Introducción

Quiero contarte una historia.

No, no es la historia de cómo, en 1991, Linus Torvalds escribió la primera versión del kernel de Linux. Puedes leer esa historia en montones de libros sobre Linux. Tampoco voy a contarte la historia de cómo, algunos años antes, Richard Stallman comenzó el Proyecto GNU para crear un sistema operativo libre parecido a Linux, Esa también es una historia importante, pero la mayoría de los libros de Linux también la incluyen.

No, quiero contarte la historia de como tú puedes recuperar el control de tu ordenador.
Cuando, estudiando en el instituto, comencé a trabajar con ordenadores había una revolución en marcha. La invención del microprocesador había hecho posible para la gente normal, como tú y yo, tener un ordenador. Es difícil para mucha gente hoy imaginar cómo era el mundo cuando sólo las grandes empresas y grandes gobiernos manejaban ordenadores. Digamos que no se podía hacer mucho más.

Hoy, el mundo es muy diferente. Los ordenadores están en todas partes, desde pequeños relojes de pulsera a gigantescos centros de datos. Además de ordenadores repartidos por todas partes también tenemos redes conectándolos. Ésto ha creado una maravillosa nueva era de empoderamiento personal y libertad creativa, pero en las dos últimas décadas algo más ha sucedido. Algunas grandes corporaciones han impuesto su control sobre la mayoría de ordenadores del mundo y han decidido lo que puedes o no puedes hacer con ellos. Afortunadamente, gente de todo el mundo está haciendo algo al respecto. Están luchando por mantener el control de sus ordenadores escribiendo su propio software. Están construyendo Linux.

Mucha gente habla de "libertad" con respecto a Linux, pero no creo que la mayoría de la gente sepa que significa esta libertad en realidad. Libertad es el poder de decidir lo que tu ordenador hace, y la única forma de tener esta libertad es saber que está haciendo tu ordenador. La libertad es un ordenador que no tiene secretos, en el que todo puede saberse si te interesa averiguarlo.

¿Por qué usar la línea de comandos?

¿Te has dado cuenta alguna vez que, en las películas, cuando el "súper hacker" - ya sabes, el tío que puede entrar en la computadora militar ultra-segura en menos de treinta segundos - se sienta en el ordenador, nunca toca un ratón? Es por que los que hacen las películas saben que nosotros, como seres humanos, instintivamente sabemos que la única forma de hacer algo en un ordenador ¡es escribiendo en el teclado!

La mayoría de los usuarios de ordenadores de hoy sólo están familiarizados con la *interfaz gráfica de usuario* o GUI (del inglés *graphical user interface*) y los vendedores y los expertos les han enseñado que la *interfaz de línea de comandos* o CLI (del inglés *command line interface*) es una cosa espantosa del pasado. Es una pena, porque una buena interfaz de línea de comandos es una maravillosa y expresiva forma de comunicarse con el ordenador, muy parecida a lo que el lenguaje escrito es para los seres humanos. Se ha dicho que "las interfaces gráficas de usuario hacen fáciles las tareas fáciles, mientras que las interfaces de línea de comandos hacen posibles las tareas difíciles" y eso es muy cierto aún hoy.

Desde que Linux fue desarrollado desde la familia de sistemas operativos Unix, comparte la misma rica herencia de herramientas de línea de comandos que Unix. Unix saltó a la fama en los primeros años ochenta (aunque fue desarrollado una década antes), antes de que se extendiera la adopción de las interfaces gráficas de usuario, y por eso, se desarrolló una amplia interfaz de línea de comandos

en su lugar.

¿De qué trata este libro?

Este libro es una visión amplia sobre "vivir" en la línea de comandos de Linux. Al contrario que algunos libros que se concentran en un sólo programa, como el programa shell o interfaz, `bash`, este libro tratará de explicar como utilizar la interfaz de línea de comandos en un sentido amplio. ¿cómo funciona? ¿qué puede hacer? ¿cuál es la mejor forma de usarla?

Éste no es un libro sobre la administración de sistemas Linux. Mientras que cualquier debate serio sobre la línea de comandos lleva inevitablemente a asuntos de administración de sistemas, este libro sólo tratará algunas cuestiones de administración. Sin embargo, preparará al lector para un estudio posterior proporcionando unos cimientos sólidos en el uso de la línea de comandos, una herramienta esencial para cualquier tarea seria de administración de sistemas.

Este libro está muy centrado en Linux. Muchos otros libros tratan de ser más atractivos incluyendo otras plataformas como Unix genérico y OS X. Al hacerlo, se convierten en una presentación de tópicos generales. Este libro, por otra parte, sólo cubre distribuciones Linux actuales. El noventa y cinco por ciento del contenido es útil para usuarios de otros sistemas parecidos a Linux, pero este libro está muy enfocado a las interfaces de línea de comandos de Linux modernas.

¿Quién debería leer este libro?

Este libro es para nuevos usuarios de Linux que hayan migrado desde otras plataformas. Lo más probable es que seas un "usuario avanzado" de alguna versión de Microsoft Windows. Quizás tu jefe te ha dicho que administres un servidor Linux, o quizás eres sólo un usuario de escritorio que está cansado de todos los problemas de seguridad y quieres darle una oportunidad a Linux. Eso está bien. Todos sois bienvenidos.

Dicho esto, no hay atajos para dominar Linux. Aprender la línea de comandos es un reto y requiere un esfuerzo real. No es que sea muy duro, más bien es muy *extenso*. El sistema Linux medio tiene literalmente *miles* de programas que puedes usar en la linea de comandos. Considérate avisado; aprender la linea de comandos no es un esfuerzo puntual.

Por otro lado, aprender la línea de comandos de Linux es extremadamente provechoso. Si crees que eres un "usuario avanzado" ahora, espera. No sabes cual es su poder real - todavía. Y al contrario que otras habilidades informáticas, el conocimiento de la linea de comandos es para mucho tiempo. Las habilidades aprendidas hoy serán útiles todavía dentro de diez años. La línea de comandos ha sobrevivido al paso del tiempo.

Se asume también que no tienes experiencia programando, pero no te preocupes, te iniciaremos en ese camino también.

¿Qué hay en este libro?

Este material está presentado en un orden cuidadosamente escogido, como si tuvieras un tutor sentado junto a ti para guiarte. Muchos autores tratan esta materia de una forma "sistemática", lo que tiene sentido desde la perspectiva del autor, pero puede ser confuso para los nuevos usuarios.

Otro objetivo es familiarizarte con la forma de pensar de Unix, que es distinta de la forma de pensar de Windows. A lo largo de nuestro camino, tomaremos algunos desvíos para ayudarte a comprender por qué algunas cosas funcionan de un modo concreto y por qué lo hacen así. Linux no

es sólo una pieza de software, es también una pequeña parte de la gran cultura Unix, que tiene su propio lenguaje e historia. Podría hacer alguna reflexión también.

Este libro está dividido en cuatro partes, cada una cubre un aspecto de la experiencia con la línea de comandos:

- **Parte 1 – Aprendiendo el Shell** comienza nuestra exploración del lenguaje básico de la línea de comandos incluyendo cosas como la estructura de los comandos, el sistema de navegación de ficheros, edición en línea de comandos, y encontrar ayuda y documentación sobre los comandos.
- **Parte 2 – Configuración y Entorno** cubre la edición de ficheros de configuración que controlan el funcionamiento del ordenador desde la línea de comandos.
- **Parte 3 – Tareas comunes y herramientas esenciales** explora muchas de las tareas ordinarias que se realizan normalmente desde la línea de comandos. Los sistemas operativos como-Unix (similares a Unix), uno de ellos es Linux, contienen muchos programas de línea de comandos “clásicos” que se usan para realizar grandes operaciones con datos.
- **Parte 4 – Escribiendo Scripts de Shell** presenta la programación en Shell, una técnica, rudimentaria pero fácil de aprender, para automatizar muchas tareas comunes de computación. Aprendiendo programación en Shell, te familiarizarás con conceptos que pueden ser aplicados a muchos otros lenguajes de programación.

¿Cómo leer este libro?

Empieza por el principio del libro y síguelo hasta el final. No está escrito como un trabajo de referencia, es más como una historia que tiene un inicio, una cumbre y un desenlace.

Prerrequisitos

Para usar este libro, todo lo que necesitas es una instalación de Linux funcionando. Puedes conseguirlo de cualquiera de estas dos formas:

1. Instala Linux en un ordenador (no tiene que ser muy potente). No importa la distribución que elijas, aunque la mayoría de las personas hoy en día empiezan con Ubuntu, Fedora o OpenSuse. Si estás en duda, prueba primero Ubuntu. Instalando una distribución de Linux moderna puede ser ridículamente fácil o ridículamente difícil dependiendo de tu hardware. Recomiendo un ordenador de hace un par de años y que tenga al menos 256 megabytes de RAM y 6 gigabytes libres de disco duro. Evita portátiles y redes inalámbricas si es posible, ya que suelen ser más difíciles de hacerlas funcionar. *(Nota del traductor: estos requisitos se refieren a la fecha de edición del libro, en la actualidad se necesita una hardware algo más potente.)*
2. Utiliza un “Live CD”. Una de las cosas interesantes que puedes hacer con muchas distribuciones Linux es ejecutarlas directamente desde un CDROM (o una unidad USB) sin instalar nada. Sólo entra en la configuración de tu BIOS y configura tu ordenador para que “Arranque desde el CDROM”, introduce el Live CD, y reinicia. Usar un Live CD es una gran forma de probar la compatibilidad de un ordenador antes de instalarlo. La desventaja de usar un Live CD es que podrá ser mucho más lento que un Linux instalado en el disco duro. Ubuntu y Fedora (además de otros) tienen versiones en Live CD.

Independientemente de como instales Linux, tendrás que tener ocasionalmente privilegios de súper-usuario (p.ej. administrativos) para llevar a cabo las lecciones de este libro.

Cuando tengas una instalación funcionando, empieza leyendo y siguiendo el libro con tu propio ordenador. La mayoría del material de este libro es “manos a la obra”, así que ¡siéntate y empieza a

teclear!

¿Por qué no lo llamo "GNU/Linux"?

En algunos ámbitos, es políticamente correcto llamar al sistema operativo Linux "Sistema Operativo GNU/Linux". El problema con "Linux" es que no hay una manera correcta del todo de nombrarlo, porque ha sido escrito por mucha gente distinta en un esfuerzo de desarrollo enorme y distribuido. Técnicamente hablando, Linux es el nombre del kernel (núcleo) del sistema operativo, nada más. Por supuesto, el kernel es muy importante, ya que hace que el sistema operativo funcione, pero no es suficiente para formar un sistema operativo completo.

Richard Stallman, es el genio-filósofo que fundó el Free Software Movement (Movimiento por el Software Libre), comenzó la Free Software Foundation (Fundación por el Software Libre), formó el Proyecto GNU, escribió la primera versión del Compilador GNU C (GCC: GNU C Compiler), creó la Licencia Pública General GNU (GPL: General Public License), etc., etc., etc. Él *insiste* en que se le llama "GNU/Linux" para reflejar apropiadamente las contribuciones del Proyecto GNU. Mientras que el Proyecto GNU es anterior al kernel de Linux, y las contribuciones hechas por el proyecto son muy dignas de reconocimiento, colocarlo en el nombre es injusto para todos aquellos otros que hayan hecho contribuciones significantes. Además, pienso que "Linux/GNU" sería más preciso técnicamente ya que el kernel arranca primero y todo lo demás funciona sobre él.

Popularmente, "Linux" se refiere al kernel y todo el software libre y de código abierto que viene en cualquier distribución de Linux; o sea, todo el ecosistema Linux, no solo los componentes GNU. El mercado de sistemas operativos parece que prefieren nombres de una sola palabra como DOS, Windows, Solaris, Irix, AIX. Yo he elegido usar el formato popular. Si, de todas formas, prefieres utilizar "GNU/Linux", por favor haz un "buscar y reemplazar" mentalmente mientras lees este libro. No me importará.

Agradecimientos

Quiero dar las gracias a las siguientes personas, quienes me ayudaron a hacer posible este libro:

Jenny Watson, Acquisitions Editor en Wiley Publishing quien en primer lugar me sugirió que escribiera un libro sobre scripts de shell.

John C. Dvorak, conocido y respetado columnista. En un episodio de su vídeo podcast, "Cranky Geeks," Mr. Dvorak describió el proceso de escribir: "Cielos. Escribe 200 palabras al día y en un año tendrás una novela." Este consejo me llevó a escribir una página al día hasta que tuve un libro.

Dmitri Popov escribió un artículo en Free Software Magazine titulado, "Creating a book template with Writer," que me inspiró a usar OpenOffice.org Writer para maquetar el texto. Al final resultó que funcionó maravillosamente.

Mark Polesky realizó una extraordinaria revisión del texto.

Jesse Becker, Tomasz Chrzczonowicz, Michael Levin, Spence Miner también leyeron y revisaron partes del texto.

Karen M. Shotts contribuyó con muchas horas, puliendo mi supuesto Inglés, editando el texto.

Y finalmente, los lectores de LinuxCommand.org, que me han enviado muchos amables correos. Sus ánimos me dieron la idea de que ¡Realmente merecía la pena!

¡Necesito tu ayuda!

Este libro es un proyecto en marcha, como muchos proyectos de opensource. Si encuentras un error técnico, mándame un correo a:

bshotts@users.sourceforge.net

Vuestros cambios y sugerencias aparecerán en futuras ediciones.

(Nota del traductor: para errores de traducción podéis utilizar los comentarios del blog, gracias)

¿Qué hay nuevo en The Second Internet Edition?

Esta versión de The Linux Command Line ha sufrido algún pulido adicional y modernización. En particular, se asume que la versión 4.x del `bash` es la versión estándar y el texto ha sido actualizado para reflejarlo. El número de capítulos en The Second Internet Edition ahora se corresponden con los de la edición No Starch Press. También he arreglado algunos bugs ;-).

Agradecimiento especial para las siguientes personas que me hicieron valiosos comentarios en la primera edición: Adrian Arpidez, Hu Bo, Heriberto Cantú, Joshua Escamilla, Bruce Fowler, Ma Jun, Seth King, Mike O'Donnell, Parviz Rasoulipour, Gabriel Stutzman, and Christian Wuethrich.

Para saber más

- Os dejo algunos artículos de Wikipedia sobre la gente famosa que aparecen en este capítulo:

http://en.wikipedia.org/wiki/Linus_Torvalds
http://en.wikipedia.org/wiki/Richard_Stallman

- La Free Software Foundation y el GNU Project:

http://en.wikipedia.org/wiki/Free_Software_Foundation
http://www.fsf.org
http://www.gnu.org

- Richard Stallman ha escrito mucho sobre el asunto del nombre“GNU/Linux”:

http://www.gnu.org/gnu/why-gnu-linux.html
http://www.gnu.org/gnu/gnu-linux-faq.html#tools

Colofón

Este libro fue originalmente escrito utilizando OpenOffice.org Writer con las fuentes Liberation Serif y Sans en un Dell Inspiron 530N, configurado de fábrica con Ubuntu 8.04. La versión PDF de este texto fue generada directamente con OpenOffice.org Writer. The Second Internet Edition fue producida en el mismo ordenador utilizando LibreOffice Writer en Ubuntu 12.04.

¿Qué es el Shell?

Cuando hablamos de la línea de comandos, realmente nos estamos refiriendo al shell. El *shell* es un programa que coge los comandos del teclado y los pasa al sistema operativo para ejecutarlos. Casi

todas las distribuciones Linux contienen un programa shell del Proyecto GNU llamado `bash`. El nombre “bash” es un acrónimo de “Bourne Again SHell”, una referencia al hecho de que `bash` es un sustituto mejorado de `sh`, el programa Shell original de Unix escrito por Steve Bourne.

Emuladores de Terminal

Cuando utilizamos una interfaz gráfica de usuario, necesitamos otro programa llamado *emulador de terminal* para interactuar con el shell. Si buscamos en nuestros menús de escritorio, probablemente encontraremos uno. KDE usa `konsole` y GNOME usa `gnome-terminal`, aunque es probable que se llame simplemente “terminal” en nuestro menú. Hay muchos otros emuladores de terminal disponibles para Linux, pero todos hacen básicamente lo mismo; nos dan acceso al shell. Seguramente desarrollarás preferencia por uno u otro según el número de “extras” que tenga.

Tus primeras pulsaciones en el teclado

Pues empecemos. ¡Arranca el emulador de terminal! Una vez que se abre, deberíamos ver algo como esto:

```
[me@linuxbox ~]$
```

Esto se llama *shell prompt* y aparecerá donde quiera que el shell esté listo para aceptar entradas. Aunque puede variar algo en apariencia según la distribución, normalmente incluirá tu *nombredeusuario@nombredetumaquina*, seguido del directorio de trabajo actual (veremos esto dentro de un poco) y un signo de dólar.

Si el último carácter del prompt es un signo de libra (“#”) en lugar de un signo de dólar, la sesión de terminal tiene privilegios de *superusuario*. Esto significa que o hemos iniciado la sesión como usuario root o hemos seleccionado un emulador de terminal que proporciona privilegios (administrativos) de superusuario.

Asumiendo que todo va bien por ahora, intentemos teclear algo. Introduce letras sin sentido en el prompt, como ésto:

```
[me@linuxbox ~]$ kaekfjaeifj
```

Como el comando no tiene sentido, el shell nos dirá que nos da otra oportunidad:

```
bash: kaekfjaeifj: command not found

[me@linuxbox ~]$
```

Historial de Comandos

Si pulsamos la tecla de la flecha hacia arriba, veremos que el comando anterior “kaekfjaeifj” reaparece tras el prompt. Esto se llama Historial de Comandos. La mayoría de distribuciones Linux recuerdan los últimos 500 comandos por defecto. Pulsa la tecla de la flecha hacia abajo y el comando anterior desaparece.

Movimiento del Cursor

Recupera el comando anterior con la tecla de la flecha hacia arriba de nuevo. Ahora prueba las teclas de las flechas hacia la izquierda y la derecha. ¿Ves cómo podemos colocar el cursor en cualquier lugar de la línea de comandos? Esto hace que la edición de comandos sea fácil.

Unas palabras sobre el Ratón y el Foco

Aunque que en el shell todo se hace con el teclado, puedes usar un ratón con tu emulador de terminal. Hay un mecanismo incluido en el X Window System (Sistema de Ventanas X, el motor subyacente que hace que el GUI funcione) que soporta una técnica de copiado y pegado rápidos. Si seleccionas un texto presionando el botón izquierdo del ratón y arrastrando el ratón sobre él (o haciendo doble clic en una palabra), se copia en un buffer mantenido por X. Presionar el botón central hace que el texto se pegue en la localización del cursor. Pruébalo.

Nota: No trates de usar `Ctrl-c` y `Ctrl-v` para copiar y pegar en una ventana de terminal. No funciona. Estos códigos de control tienen diferentes significados para el shell y fueron asignados muchos años antes de Microsoft Windows.

Tu entorno gráfico de escritorio (seguramente KDE o GNOME), en un esfuerzo por comportarse como Windows, probablemente tiene su *política de foco* configurada como “clic para foco”. Esto significa que para que una ventana tenga foco (se ponga activa) necesitas hacer clic en ella. Esto es contrario al comportamiento tradicional de X de “el foco sigue al ratón” que significa que una ventana toma el foco sólo pasando el ratón sobre ella. La ventana no vendrá al primer plano hasta que no hagas clic en ella pero estará lista para recibir entradas. Configurando la política de foco como “el foco sigue al ratón” hará que la técnica de copiado y pegado sea aún más cómoda. Dale una oportunidad si puedes (algunos entornos de escritorio como Unity de Ubuntu ya no lo soportan). Creo que si le das una oportunidad lo preferirás. Encontrarás esta configuración en el programa de configuración de tu gestor de ventanas.

Prueba algunos comandos sencillos

Ahora que hemos aprendido a teclear, probemos algunos comandos sencillos. El primero es `date`. Este comando muestra la hora y fecha actual.

```
[me@linuxbox ~]$ date
Thu Oct 25 13:51:54 EDT 2007
```

Un comando relacionado es `cal` que, por defecto, muestra un calendario del mes actual.

```
[me@linuxbox ~]$ cal
    October 2007
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

Para ver la cantidad actual de espacio disponible en tus unidades de disco, escribe `df`:

```
[me@linuxbox ~]$ df
Filesystem 1K-blocks     Used Available Use% Mounted on
/dev/sda2   15115452  5012392   9949716  34% /
/dev/sda5   59631908 26545424  30008432  47% /home
/dev/sda1     147764    17370    122765  13% /boot
tmpfs       256856          0    256856   0% /dev/shm
```

De la misma forma, para mostrar la cantidad de memoria libre, escribe el comando `free`.

```
[me@linuxbox ~]$ free
```

```
             total       used    free  shared buffers  cached
Mem:        513712     503976    9736       0    5312  122916
-/+ buffers/cache:     375748  137964
Swap:      1052248     104712  947536
```

Finalizando una sesión de terminal

Podemos finalizar una sesión de terminal cerrando la ventana del emulador de terminal, o escribiendo el comando `exit` en el prompt del shell:

```
[me@linuxbox ~]$ exit
```

La consola tras el telón

Aunque no tengamos ningún emulador de terminal funcionando, algunas sesiones de terminal continúan funcionado detrás del escritorio gráfico. Llamados *terminales virtuales* o *consolas virtuales*, podemos acceder a estas sesiones en la mayoría de las distribuciones Linux pulsando `Ctrl-Alt-F1` a `Crtl-Alt-F6`. Cuando accedemos a una sesión, presenta un prompt de acceso en el que podemos introducir nuestro usuario y contraseña. Para cambiar de una consola virtual a otra, presiona `Alt` y `F1-F6`. Para volver al escritorio gráfico, pulsa `Alt-F7`.

Resumiendo

Para empezar nuestro viaje, hemos presentado el shell y hemos visto la línea de comandos por primera vez y aprendido como empezar y terminar una sesión de terminal. También hemos visto como enviar algunos comandos sencillos y realizar una pequeña edición en la línea de comandos. No da tanto miedo ¿no?

Para saber más

Para aprender más sobre Steve Bourne, el padre del Bourne Shell, lee este artículo de Wikipedia: http://en.wikipedia.org/wiki/Steve_Bourne

Aquí hay un artículo sobre el concepto de los shells en informática: http://en.wikipedia.org/wiki/Shell_(computing)

Navegación

Lo primero que necesitamos aprender (además de teclear) es como navegar por el sistema de archivos en nuestro sistema Linux. En este capítulo presentaremos los siguientes comandos:

- `pwd` – Muestra el nombre del directorio de trabajo actual
- `cd` – Cambia de directorio
- `ls` – Muestra el contenido del directorio

Entendiendo el árbol del sistema de archivos

Como Windows, un sistema operativo "como-Unix", como es Linux, organiza sus archivos en lo que se llama una *estructura jerárquica de directorios*. Ésto significa que están organizados una estructura de directorios con forma de árbol (algunas veces se llaman carpetas en otros sistemas), que pueden contener archivos u otros directorios. El primer directorio en el sistema de archivos se llama el *directorio raíz* (root directory). El directorio raíz contiene archivos y subdirectorios, que

contienen más archivos y más subdirectorios y así sucesivamente.

Nota que al contrario que Windows, que tiene un árbol del sistema de directorios separado para cada dispositivo de almacenamiento, los sistemas como-Unix, como es Linux, siempre tienen un único árbol de sistema, independientemente de cuantas unidades o dispositivos de almacenamiento están conectados al ordenador. Los dispositivos de almacenamiento están conectados (o más correctamente, *montados*) en varios puntos del árbol según la voluntad del *administrador del sistema*, la persona (o personas) responsables del mantenimiento del sistema.

El directorio de trabajo actual

La mayoría de nosotros, probablemente, estamos familiarizados con un gestor de archivos gráfico que representa el árbol del sistema de directorios como en la Figura 1. Fíjate que el árbol, normalmente, aparece colgando bocabajo, o sea, con la raíz arriba y las ramas hacia abajo.

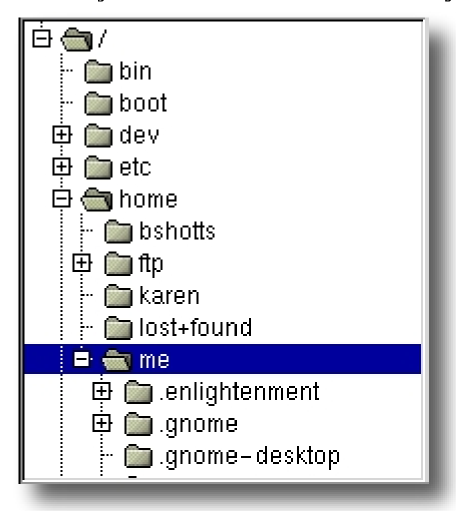


Figura 1: El árbol del sistema de ficheros como se ve en un gestor de archivos gráfico

Sin embargo, la línea de comandos no tiene imágenes, así que para navegar por el árbol del sistema de ficheros tenemos que pensar de una forma diferente.

Imagina que el sistema de ficheros es como un laberinto con forma de árbol bocabajo y que podemos estar dentro de él. En un momento dado, estamos dentro de un directorio en concreto y podemos ver los archivos contenidos en él, y la ruta hacia el directorio que está sobre nosotros (llamado *directorio padre*) y los subdirectorios debajo de nosotros. Éste directorio, en el cual estamos, se llama *directorio de trabajo actual*. Para mostrar el directorio de trabajo actual, utilizamos el comando `pwd` (print working directory):

```
[me@linuxbox ~]$ pwd

/home/me
```

La primera vez que accedemos al sistema (o abrimos una sesión del emulador de terminal) nuestro directorio de trabajo actual es nuestro *directorio home* (se podría traducir como directorio hogar o directorio de usuario). Cada cuenta de usuario tiene su propio directorio home y es el único lugar donde un usuario normal (sin privilegios) puede escribir archivos.

Listando los contenidos de un directorio

Para listar los archivos y directorios en el directorio de trabajo actual, usamos el comando `ls`.

```
[me@linuxbox ~]$ ls
Desktop Documents Music Pictures Public Templates Videos
```

En realidad, podemos usar el comando `ls` para listar el contenido de cualquier directorio, no sólo el directorio de trabajo actual, y hay muchas otras cosas curiosas que puede hacer también. Pasaremos más tiempo con `ls` en el próximo capítulo.

Cambiando el directorio de trabajo actual

Para cambiar el directorio de trabajo actual (aquel en el que estamos en nuestro laberinto con forma de árbol) utilizamos el comando `cd`. Par hacer ésto, escribe `cd` seguido de la *ruta* del directorio de trabajo que queramos. Una ruta es el camino de tomamos a través de las ramas del árbol para llegar al directorio que queremos. Las rutas pueden ser especificadas de dos formas; como *rutas absolutas* o como *rutas relativas*. Veremos las rutas absolutas primero.

Rutas Absolutas

Una ruta absoluta comienza con el directorio raíz y sigue rama tras rama hasta que la ruta al directorio o archivo que queremos está completa. Por ejemplo, hay un directorio en nuestro sistema en el que la mayoría de los programas de sistema están instalados. La ruta de dicho directorio es `/usr/bin`. Ésto significa que en el directorio raíz (representado por la barra inclinada en la ruta) hay un directorio llamado "usr" que contiene un directorio llamado "bin".

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls
...Listing of many, many files ...
```

Ahora podemos ver que hemos cambiado el directorio de trabajo a /usr/bin y que está lleno de archivos. ¿Ves como ha cambiado el prompt del shell? Por defecto, normalmente está configurado para mostrar automáticamente el nombre del directorio de trabajo.

Rutas Relativas

Mientras que una ruta absoluta empieza desde el directorio raíz y sigue hasta su destino, una ruta relativa comienza en el directorio de trabajo. Para hacer ésto, utiliza un par de símbolos especiales que representan posiciones relativas en el árbol del sistema de directorios. Estos símbolos son "." (punto) y ".." (punto punto).

El símbolo "." se refiere al directorio de trabajo y el símbolo ".." se refiere al directorio padre del directorio de trabajo. Veamos como funciona. Cambia el directorio de trabajo a `/usr/bin` de nuevo:

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Ok, ahora le diremos que queremos cambiar el directorio de trabajo al padre de `/usr/bin` que es `/usr`. Podríamos hacerlo de dos formas diferentes. Tanto con una ruta absoluta:

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

Como, con una ruta relativa:

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

Dos métodos diferentes con idénticos resultados. ¿Cuál usamos? ¡El que requiera teclear menos!

Igualmente, podemos cambiar el directorio de trabajo de /usr a /usr/bin de dos formas distintas. O usando una ruta absoluta:

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

O con una ruta relativa:

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Ahora, hay algo importante que debo señalar aquí. En la mayoría de los casos, puedes omitir el “./”. Está implícito. Escribir:

```
[me@linuxbox usr]$ cd bin
```

hace lo mismo. En general, si no especificas una ruta a algo, se asumirá el directorio de trabajo.

Algunos atajos útiles

En la Tabla 2-1 vemos algunas formas útiles para cambiar rápidamente el directorio de trabajo actual.

Tabla 2-1: Atajos de cd

Atajo	Resultado
cd	Cambia el directorio de trabajo a tu directorio home
cd -	Cambia el directorio de trabajo al anterior directorio de trabajo
cd ~*nombre_de_usuario*	Cambia el directorio de trabajo al directorio home de *nombre_de_usuario*. Por ejemplo, **cd ~bob** cambiará el directorio al directorio home del usuario “bob”.

Aspectos importantes sobre nombres de archivo

1. Los nombres de archivo que empiezan con un punto están ocultos. Ésto sólo significa que `ls` no los listará a menos que digamos `ls -a`. Cuando se creó tu cuenta, varios archivos ocultos fueron colocados en tu directorio home para configurar cosas de tu cuenta. Más adelante veremos más a fondo algunos de estos archivos para ver como puedes personalizar tu entorno. Adicionalmente, algunas aplicaciones colocan su configuración y sus ajustes en tu directorio home como archivos ocultos.
2. Los nombres de archivo y los comandos en Linux, como en Unix, son sensibles a las mayúsculas. Los nombres de archivo “File1” y “file1” se refieren a archivos diferentes.
3. Linux no tiene el concepto de “extensión de archivo” como algunos otros sistemas

operativos. Puedes nombrar un archivo de la forma que quieras. Los contenidos y/o propósito de un archivo se determina por otros medios. Aunque los sistemas operativos como-Unix no usan extensiones de archivo para determinar el contenido/propósito de un archivo, algunas aplicaciones sí lo hacen.

4. Aunque Linux soporta nombres de archivo largos que pueden contener espacios y signos de puntuación, limita los signos de puntuación en los nombres de archivo que crees al punto, el guión y el guión bajo. *Lo más importante, no incluyas espacios en los nombres de archivo*. Si quieres representar espacios entre palabras en un nombre de archivo, usa guiones bajos. Te lo agradecerás más tarde.

Resumiendo

En este capítulo hemos visto como el shell trata la estructura de directorios del sistema. Hemos aprendido sobre las rutas absolutas y relativas y los comandos básicos que se usan para moverse por dicha estructura. En el próximo capítulo utilizaremos este conocimiento para darnos una vuelta por un sistema Linux moderno.

Explorando el sistema

Ahora que sabemos como movernos por el sistema de archivos, es hora para un tour guiado por nuestro sistema Linux. Antes de empezar sin embargo, vamos a aprender algunos comandos más que serán útiles en nuestro viaje:

- `ls` – Lista los contenidos de un directorio
- `file` – Muestra el tipo de archivo
- `less` – Muestra el contenido del archivo

Más diversión con ls

El comando `ls` es probablemente el más usado, y por una buena razón. Con él, podemos ver los contenidos de un directorio y determinar varios atributos importantes de los archivos y directorios. Como hemos visto, podemos usar simplemente `ls` para ver una lista de los archivos y subdirectorios contenidos en el directorio de trabajo actual:

```
[me@linuxbox ~]$ ls
Desktop Documents Music Pictures Public Templates Videos
```

Además del directorio de trabajo actual, podemos especificar el directorio a listar, de esta forma:

```
me@linuxbox ~]$ ls /usr
bin  games    kerberos    libexec    sbin    src
etc  include  lib         local      share   tmp
```

O incluso especificar múltiples directorios. En este ejemplo listaremos el directorio home del usuario (simbolizado por el carácter “~”) y el directorio `/usr`:

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
Desktop Documents Music Pictures Public Templates Videos
/usr:
bin games   kerberos  libexec  sbin  src
etc include lib       local    share tmp
```

También podemos cambiar el formato de la salida para que nos muestre más detalles:

```
[me@linuxbox ~]$ ls -l
total 56
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Desktop
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Documents
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Music
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Pictures
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Public
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Templates
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Videos
```

Añadiendo "`-l`" al comando, cambiamos la salida al formato largo.

Opciones y argumentos

Ésto nos ofrece un aspecto muy importante sobre como funcionan la mayoría de los comandos. A los comandos, a menudo, le siguen una o más opciones que modifican su comportamiento, y adicionalmente, por uno o más argumentos, los elementos sobre los que el comando actúa. Así que la mayoría de los comandos aparecen como algo así:

comando -opciones argumentos

La mayoría de los comandos usan opciones que consisten en un sólo carácter precedido por un guión, por ejemplo, "`-l`", pero muchos comandos, incluidos los del Proyecto GNU, también soportan *opciones largas*, consistentes en una palabra precedida por dos guiones. Además, muchos comandos permiten múltiples opciones cortas enlazadas. En este ejemplo, al comando `ls` se le han dado dos opciones, la opción "`l`" para mostrar la salida en formato largo, y la opción "`t`" para ordenar el resultado por la hora de modificación de los archivos.

```
[me@linuxbox ~]$ ls -lt
```

Añadiremos la opción larga "`--reverse`" para invertir el orden de la lista:

```
[me@linuxbox ~]$ ls -lt --reverse
```

Fíjate que las opciones de los comandos, como los nombres de archivo en Linux, son sensibles a las mayúsculas.

El comando `ls` tiene un gran número de posibles opciones. Las más comunes están listadas en la Tabla 3-1.

Tabla 3-1. Opciones comunes de ls

Opción	Opción larga	Descripción
`-a`	`--all`	Lista todos los archivos, incluso aquellos cuyo nombre empieza con un punto, que normalmente no se listan (p.ej. Ocultos)
`-A`	`--almost-all`	Como la opción `-a` anterior, salvo que no lista . (el directorio actual) y .. (el directorio padre).
`-d`	`--directory`	Por defecto, si especificamos un directorio, `ls` listará los contenidos del directorio y no el directorio en sí. Usa esta opción junto con la opción `-l` para ver detalles del directorio además de su contenido.

Opción	Opción larga	Descripción
`-F`	`--classify`	Esta opción añadirá un carácter indicador al final de cada nombre listado. Por ejemplo, una “/” es que el nombre es un directorio.
`-h`	`--human-readable`	En los listados en formato largo, muestra el tamaño de los archivos en un formato comprensible, en lugar de en bytes.
`-l`		Muestra los resultados en formato largo
`-r`	`--reverse`	Muestra los resultados en orden inverso. Normalmente, `ls` muestra los resultados en orden alfabético ascendente.
`-S`		Ordena los resultados por tamaño,
`-t`		Ordena por hora de modificación.

Un vistazo más profundo al formato largo

Como vimos antes, la opción “`-l`” hace que `ls` muestre sus resultados en formato largo. Este formato contiene gran cantidad de información útil. Aquí está el directorio `Examples` de un sistema Ubuntu:

```
-rw-r--r-- 1 root root 3576296 2007-04-03 11:05 Experience
ubuntu.ogg
-rw-r--r-- 1 root root 1186219 2007-04-03 11:05 kubuntu-
leaflet.png
-rw-r--r-- 1 root root 47584 2007-04-03 11:05 logo-
Edubuntu.png
-rw-r--r-- 1 root root 44355 2007-04-03 11:05 logo-
Kubuntu.png
-rw-r--r-- 1 root root 34391 2007-04-03 11:05 logo-
Ubuntu.png
-rw-r--r-- 1 root root 32059 2007-04-03 11:05 oo-cd-
cover.odf
-rw-r--r-- 1 root root 159744 2007-04-03 11:05 oo-
derivatives.doc
-rw-r--r-- 1 root root 27837 2007-04-03 11:05 oo-
maxwell.odt
-rw-r--r-- 1 root root 98816 2007-04-03 11:05 oo-trig.xls
-rw-r--r-- 1 root root 453764 2007-04-03 11:05 oo-
welcome.odt
-rw-r--r-- 1 root root 358374 2007-04-03 11:05 ubuntu
Sax.ogg
```

Veamos los diferentes campos de uno de los archivos y examinemos su significado:

Tabla 3-2: Campos del listado largo con ls

Campo	Significado
`-rw-r—r--`	Permisos de acceso al archivo. El primer carácter indica el tipo de archivo. Entre los diferentes tipos, un guión medio significa un archivo normal, mientras que una “d” indica un directorio. Los siguientes tres caracteres son los permisos de acceso para el propietario del archivo, los tres siguientes los de los miembros del grupo del archivo, y lo tres últimos los del resto de usuarios. El significado completo de esto lo trataremos en el

	Capítulo 9 – Permisos.
`1`	Numero de enlaces fuertes del archivo. Veremos el tema de los enlaces más adelante en este capítulo.
`root`	El nombre del propietario del archivo.
`root`	El nombre del grupo del propietario del archivo.
`32059`	Tamaño del archivo en bytes.
`2007-04-03 11:05`	Fecha y hora de la última modificación del archivo.
`oo-cd-cover.odf`	Nombre del archivo.

Averiguando el tipo de archivo con file

Al explorar el sistema será útil saber que contienen los archivos. Para hacerlo utilizaremos el comando `file` para determinar el tipo de archivo. Como vimos antes, los nombres de archivo en Linux no necesitan reflejar el contenido del archivo. Mientras que un nombre de archivo como "picture.jpg" se espera que contenga una imagen comprimida JPEG, ésto no tiene porqué ser así en Linux. Podemos invocar el comando `file` de la siguiente forma:

file *nombre_del_archivo*

Cuando invocamos, el comando `file` mostrará una breve descripción del contenido del archivo. Por ejemplo:

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01
```

Hay muchos tipos de archivos. De hecho, una de las ideas comunes en los sitemas operativos como-Unix es que "todo es un archivo". Según avancemos en nuestras lecciones, veremos como de cierta es esta expresión.

Mientras que muchos de los archivos en tu sistema son familiares, por ejemplo MP3 y JPEG, hay otros muchos que son un poco menos obvios y algunos que son un poco raros.

Viendo el contenido de un archivo con less

El comando `less` es un programa para ver archivos de texto. En nuestro sistema Linux, hay muchos archivos que contienen texto comprensible. El programa `less` proporciona una forma adecuada para examinarlos.

¿Para qué querríamos examinar archivos de texto? Porque muchos de los archivos que contienen la configuración del sistema (llamados *archivos de configuración*) está almacenados en este formato, y ser capaces de leerlos nos permite ver como funciona el sistema por dentro. Además, muchos de los programas que el sistema utiliza (llamados *scripts*) están almacenados en este formato. En capítulos posteriores, aprenderemos como editar archivos de texto para modificar configuraciones del sistema y escribiremos nuestros propios scripts, pero por ahora sólo veremos su contenido.

El comando `less` se usa de la siguiente manera:

less *nombre_del_archivo*

Una vez ejecutado, el programa `less` nos permite desplazarnos arriba y abajo a través de un

archivo de texto. Por ejemplo, para examinar el archivo que define todas las cuentas de usuario del sistema, usamos el siguiente comando:

```
[me@linuxbox ~]$ less /etc/passwd
```

Una vez que el programa `less` arranca, podemos ver el contenido del archivo. Si el archivo ocupa más de una página, podemos desplazarnos arriba y abajo. Para salir de `less`, pulsamos la tecla "q".

La siguiente tabla muestra los comandos de teclado más comunes que se usan con `less`.

Comando	Acción
Re Pag o b	Se desplaza hacia atrás una página
Av Pag o espacio	Se desplaza hacia delante una página
Flecha hacia arriba	Se desplaza hacia arriba una línea
Flecha hacia abajo	Se desplaza hacia abajo una linea
G	Se mueve al final del archivo de texto
1G o g	Se mueve al principio de un archivo de texto
/ *caracteres*	Busca hacia delante hasta la siguiente coincidencia de *caracteres*
n	Busca la siguiente coincidencia de caracteres de la búsqueda anterior
h	Muestra la pantalla de ayuda
q	Cierra *less*

¿Qué es "texto"?

Hay muchas formas de representar información en un ordenador. Todos los métodos implican definir una relación entre la información y los números que se utilizarán para representarla. Los ordenadores, después de todo, sólo entienden los números, y todos los datos son convertidos a una representación numérica.

Algunas de estas representaciones son muy complejas (como los vídeos comprimidos), mientras que otras son bastante simples. Una de las primeras y más simples es el llamado *texto ASCII*. ASCII (pronunciado "aski") es la abreviatura de American Standard Code for Information Interchange (Código Estándar Americano para Intercambio de Información). Es un esquema de codificación simple que se usó por primera vez en las maquinas de Teletipo para convertir caracteres de teclado en números.

El texto es una conversión simple uno-a-uno de texto en números. Es muy compacto. Cincuenta caracteres de texto se trasladan a cincuenta bytes de datos. Es importante comprender que el texto sólo contiene una conversión simple de caracteres a números. No es lo mismo que un documento de procesador de texto como los creados con Microsoft Word o OpenOffice.org Writer. Esos archivos, al contrario que el texto simple ASCII, contienen muchos elementos que no son texto y que sirven para describir la estructura y el formato. Los archivos de texto plano ASCII contienen sólo los

caracteres y algunos códigos de control muy rudimentarios como las etiquetas, retornos de carro y saltos de línea.

A lo largo de un sistema Linux, muchos archivos están almacenados en formato texto y hay muchas herramientas de Linux que trabajan con archivos de texto. Incluso Windows reconoce la importancia de este formato. El famoso programa NOTEPAD.EXE es un editor para archivos de texto plano ASCII.

Menos es Más (Less is More)

El programa `less` fue diseñado como un reemplazo mejorado de un programa anterior de Unix llamado `more`. El nombre less viene de un juego de palabras "menos es mas – less is more" - un lema de los arquitectos y diseñadores modernistas.

`less` entra en la clase de programas llamados "paginadores", que permiten ver fácilmente documentos de texto largos en forma de páginas. Mientras que el programa `more` sólo permitía avanzar páginas, el programa `less` permite avanzar y retroceder por las páginas y cuenta también con otras características.

Una visita guiada

La disposición del sistema de archivos en tu sistema Linux es casi como el que encontramos en otros sistemas como-Unix. El diseño está especificado en un estándar publicado llamado *Linux Filesystem Hierarchy Standard (Jerarquía Estándar del sistema de archivos de Linux)*. No todas las distribuciones Linux cumplen el estándar exactamente pero la mayoría se acercan bastante.

A continuación, vamos a dar un paseo por el sistema de archivos para darnos cuenta de lo que hace nuestro sistema linux. Ésto nos dará la oportunidad de practicar nuestras habilidades de navegación. Una de las cosas que descubriremos es que muchos de los archivos interesantes están en texto plano legible. A medida que vamos avanzando en nuestra visita guiada, prueba lo siguiente:

1. `cd` en un directorio dado
2. Lista los contenidos del directorio con `ls -l`
3. Si ves un archivo interesante, determina su contenido con `file`
4. Si parece que pudiera ser texto, trata de verlo con `less`

¡Recuerda el truco para copiar y pegar! Si utilizas ratón, puedes hacer doble clic en el nombre de un archivo para copiar y clic con el botón central para pegarlo en los comandos.

A medida que avancemos, no tengas miedo de echar un vistazo. Los usuarios normales tienen terminantemente prohibido desordenar cosas. ¡Es trabajo para los administradores de sistema! Si un comando protesta sobre algo, pasa a otra cosa. Pasa algo de tiempo echando un vistazo. El sistema es nuestro para explorarlo. Recuerda, en Linux, ¡no hay secretos!

La tabla 3-4 lista sólo algunos de los directorios que podemos explorar. ¡Eres libre de probar con más!

Tabla 3-4: Directorios que se encuentran en los Sistemas Linux

Directorio	Comentarios
`/`	El directorio raíz. Donde comienza todo.
`/bin`	Contiene binarios (programas) que tienen que estar presentes para que el sistema

	arranque y funcione.
`/boot`	Contiene el kernel de Linux, la imagen del disco de RAM inicial (para controladores necesarios en el momento del arranque), y el cargador de arranque. • `/boot/grub/grub.conf` o `menu.lst`, que se utilizan para configurar el cargador de arranque. • `/boot/vmlinuz`, el kernel de Linux.
`/dev`	Éste es un directorio especial que contiene los *nodos de dispositivo*. “Todo es un archivo” también se aplica a los dispositivos. Aquí es donde el kernel mantiene una lista de todos los dispositivos que comprende.
`/etc`	El directorio `/etc` contiene todos los archivos de configuración del sistema. También contiene una colección de scripts de shell que se ejecutan con cada servicio del sistema en el arranque. Todo en este directorio debería ser texto legible. Archivos interesantes: Como todo en `/etc` es interesante, os dejo algunos de mis favoritos de siempre: • `/etc/crontab`, un archivo que define cuando los trabajos automáticos se ejecutarán. • `/etc/fstab`, una tabla de los dispositivos de almacenamiento y sus puntos de montaje asociados. • `/etc/passwd`, una lista de las cuentas de usuario.
`/home`	En configuraciones normales, a cada usuario se le asigna un directorio en `/home`. Los usuarios normales sólo pueden escribir archivos en sus directorios home. Esta limitación protege el sistema de actividades erróneas de los usuarios.
`/lib`	Contiene archivos de librerías compartidas utilizadas por los programas del núcleo del sistema. Son similares a las DLL de Windows.
`/lost+found`	Cada partición formateada o dispositivo que usa un sistema de archivos de Linux como ext3, tendrá este directorio. Se usa en caso de una recuperación parcial causada por un evento de corrupción del sistema de archivos. A menos que algo realmente malo ocurra a tu sistema, este directorio debería permanecer vacío.
`/media`	En sistemas Linux modernos el directorio `/media` contendrá los puntos de montaje de los dispositivos extraibles como unidades USB, CD-ROMs, etc. que se montan automáticamente al introducirlos.
`/mnt`	El sistemas Linux antiguos, el directorio `/mnt` contiene los puntos de montaje de los dispositivos extraíbles que han sido montados manualmente.
`/opt`	El directorio `/opt` se usa para instalar software “opcional”. Principalmente se usa para contener productos de software comercial que podrían ser instalados en tu sistema.
`/proc`	El directorio `/proc` es especial. No es un sistema de ficheros real en el sentido de archivos almacenados en tu disco duro. En su lugar, es un sistema de archivos virtual mantenido por el kernel de Linux. Los “archivos” que contienen son como mirillas dentro del propio núcleo. Los archivos son legibles y os darán una imagen de como el kernel ve el ordenador.
`/root`	Es el directorio home para la cuenta root.
`/sbin`	Este directorio contiene binarios de sistema. Son programas que realizan tareas de sistema vitales que generalmente están reservadas al superusuario.

`/tmp`	El directorio `/tmp` está destinado a almacenamiento de temporales, ficheros provisionales creados por varios programas. Algunas configuraciones hacen que este directorio sea vaciado cada vez que el sistema se reinicia.
`/usr`	El árbol del directorio `/usr` es casi el más grande en un sistema Linux. Contiene todos los programas y archivos de soporte utilizados por los usuarios normales.
`/usr/bin`	`/usr/bin` contiene los programas ejecutables instalados en tu distribución Linux. No es raro que este directorio contenga miles de programas.
`/usr/lib`	Las librerías compartidas para los programas en `/usr/bin`.
`/usr/local`	El árbol `/usr/local` es donde están instalados los programas que no están incluidos en tu distribución pero están destinados a un uso general.
`/usr/sbin`	Contiene más programas de administración de sistema.
`/usr/share`	`/usr/share` continiene todos los datos compartidos utilizados por los programas en `/usr/bin`. Ésto incluye cosas como archivos de configuración por defecto, iconos, fondos de pantalla, archivos de sonido, etc.
`/usr/share/doc`	La mayoría de los paquetes instalados en el sistema incluirán algún tipo de documentación. En `/usr/share/doc`, encontraremos archivos de documentación ordenados por paquete.
`/var`	Con la excepción de `/tmp` y `/home`, los directorios que hemos visto hasta ahora permanecen relativamente estáticos, o sea, su contenido no cambia. El árbol del directorio `/var` es donde los datos que es probable que cambien son almacenados. Diferentes bases de datos, archivos de colas, correos de usuarios, etc. se encuentran aquí.
`/var/log`	`/var/log` contiene archivos de registro de varias actividades del sistema. Son muy importantes y deberían ser monitorizados de vez en cuando. El más útil es `/var/log/messages`. Tenga en cuenta que, por razones de seguridad, en algunos sistemas, tienes que ser superusuario para ver los archivos de registro.

Enlaces Simbólicos

Cuando miramos a nuestro alrededor, es probable que veamos un listado de directorio con una entrada como esta:

```
lrwxrwxrwx 1 root root 11 2007-08-11 07:34 libc.so.6 ->
libc-2.6.so
```

¿Ves como la primera letra del listado es "`l`" y la entrada parece tener dos nombres de archivo? Ésto es un tipo especial de archivo llamado *enlace simbólico* (también conocido como *enlace blando* o *sym-link*.) En la mayoría de los sistemas como-Unix es posible tener un archivo referenciado por múltiples nombres. Aunque el sentido de ésto no parezca obvio, es realmente una característica muy útil.

Imagina este escenario: Un programa requiere el uso de algún tipo de recurso compartido contenido en un archivo llamado "foo", pero "foo" tiene frecuentes cambios de versión. Sería bueno incluir el número de versión en el nombre del archivo para que el administrador u otras partes interesadas pudieran ver que versión de "foo" está instalada. Ésto presenta un problema. Si

cambiamos el nombre del recurso compartido, tenemos que localizar cada programa que pudiera usarlo y cambiarlo para que buscara un nuevo recurso cada vez que se instalara una nueva versión del recurso. Esto no suena nada divertido.

Aquí es donde los enlaces simbólicos nos arreglan el día. Digamos que instalamos la versión 2.6 de “foo”, que tiene el nombre de archivo “foo-2.6” y entonces creamos un enlace simbólico llamado simplemente “foo” que apunta a “foo-2.6”. Ahora todo el mundo está contento. Los programas que utilizan “foo” pueden encontrarlo y nosotros aún podemos ver que versión está instalada en realidad. Cuando llegue la hora de actualizar a “foo-2.7”, sólo añadiremos el archivo a nuestro sistema, borraremos el enlace simbólico “foo” y crearemos uno nuevo que apunte a la nueva versión. No sólo soluciona el problema del cambio de versión, también nos permite guardar ambas versiones en nuestra máquina. Imagina que “foo-2.7” tiene un bug (¡Condenados programadores!) y necesitamos recuperar la versión antigua. De nuevo, sólo borramos el enlace simbólico que apunta a la nueva versión y creamos un nuevo enlace simbólico apuntando a la versión antigua.

El directorio listado anteriormente (del directorio `/lib` de un sistema Fedora) muestra un enlace simbólico llamado “libc.so.6” que apunta a un archivo de librería compartida llamado “libc-2.6.so”. Ésto significa que los programas que busquen “libc.so.6” en realidad obtendrán el archivo “libc-2.6.so”. Aprenderemos como crear enlaces simbólicos en el próximo capítulo.

Enlaces duros

Como estamos en el tema de los enlaces, tenemos que mencionar que hay un segundo tipo de enlace llamado *enlace duro*. Los enlaces duros también permiten a los archivos tener múltiples nombres, pero lo hacen de una forma distinta. Hablaremos más sobre las diferencias entre enlaces simbólicos y duros en el próximo capítulo.

Resumiendo

Terminado nuestro tour, hemos aprendido un montón acerca de nuestro sistema. Hemos visto varios archivos y directorios y sus contenidos. Una cosa que deberías haber sacado de ésto es lo abierto que es el sistema. En Linux hay muchos archivos importantes que están en texto plano y legible. Al contrario que muchos sistemas propietarios, Linux hace que todo pueda ser examinado y estudiado.

Para saber más

- La versión completa de la *Jerarquía Estándar del Sistema de Archivos de Linux* puedes encontrarla aquí:

  http://www.pathname.com/fhs/

- Un artículo sobre la estructura de directorios de Unix y los sistemas como-Unix:

  http://en.wikipedia.org/wiki/Unix_directory_structure

- Una descripción detallada del formato de texto ASCII:

  http://en.wikipedia.org/wiki/ASCII

Manipulando archivos y directorios

Llegados a este punto, ¡estamos preparados para algo de trabajo real! En este capítulo presentaremos los siguientes comandos:

- `cp` – Copia archivos y directorios
- `mv` – Mueve/renombra archivos y directorios
- `mkdir` – Crea directorios
- `rm` – Borra archivos y directorios
- `ln` – Crea enlaces duros y simbólicos

Estos cinco comandos están entre los comandos de Linux usados más frecuentemente. Se usan para manipular tanto archivos como directorios.

Aunque, para ser francos, algunas de las tareas que realizan estos comandos se hacen más fácilmente con un gestor gráfico de archivos. Con un gestor de archivos, podemos arrastrar y soltar un archivo de un directorio a otro, cortar y pegar archivos, borrar archivos, etc. Entonces, ¿por qué utilizamos estos programas antiguos de línea de comandos?

La respuesta es poder y flexibilidad. Mientras que es fácil realizar manipulaciones simples de archivos con un gestor gráfico de archivos, las tareas complicadas pueden ser más fáciles con la línea de comandos. Por ejemplo, ¿cómo podríamos copiar todos los archivos HTML de un directorio a otro, pero sólo copiando los archivos que no existan en el directorio de destino o que sean más recientes que las versiones en el directorio de destino? Bastante difícil con un gestor de archivos. Bastante fácil con la línea de comandos:

```
cp -u *.html destination
```

Comodines

Antes de empezar a usar nuestros comandos, necesitamos hablar sobre una característica del shell que hace a estos comandos muy potentes. Como el shell utiliza mucho los nombres de archivo, ofrece caracteres especiales para ayudarnos a especificar rápidamente grupos o nombres de archivo. Estos caracteres especiales se llaman comodines. Utilizar *comodines* (también conocidos como *globbing*) te permiten seleccionar nombres de archivo basados en patrones o caracteres. La siguiente tabla lista los comodines y lo que seleccionan:

Tabla 4-1: Comodines

Comodín	Significado
`*`	Cualquier carácter
`?`	Cualquier carácter individual
`[caracteres]`	Cualquier carácter que sea miembro del grupo *caracteres*
`[!caracteres]`	Cualquier carácter que no sea miembro del grupo *caracteres*
`[[:class:]]`	Cualquier carácter que sea miembro de una clase específica

La Tabla 4-2 lista las clases de caracteres más comúnmente usadas:

Tabla 4-2: Clases de caracteres usadas más comúnmente

Clase de caracteres	Significado
`[:alnum:]`	Cualquier carácter alfanumérico

[:alpha:]	Cualquier carácter alfabético
[:digit:]	Cualquier numérico
[:lower:]	Cualquier letra minúscula
[:upper:]	Cualquier letra mayúscula

Utilizar comodines posibilita construir criterios de selección para nombres de archivo muy sofisticados. Aquí tienes algunos ejemplos de patrones y qué indican:

Tabla 4-3: Ejemplos de comodines

Patrón	Selección
*	Todos los archivos
g*	Todos los archivos que empiezan por “g”
b*.txt	Todos los archivos que empiecen por “g” seguido de cualquier carácter y terminados en “.txt”
Data???	Todos los archivos que empiezan por “Data” seguido de exactamente tres caracteres
[abc]*	Todos los archivos que empiezan por “a”, “b” o “c”
BACKUP.[0-9][0-9][0-9]	Todos los archivos que empiezan por “BACKUP.” seguido de exactamente tres números
[[:upper:]]*	Todos los archivos que empiezan por una letra mayúscula
[![:digit:]]*	Todos los archivos que no empiezan por un número
*[[:lower:]123]	Todos los archivos que terminan por una letra minúscula o por los números “1”, “2” o “3”

Los comodines pueden usarse con cualquier comando que acepte nombres de archivo como argumento, pero hablaremos más sobre ello en el Capítulo 7.

Rangos de caracteres

Si vienes de otros entornos como-Unix o has estado leyendo otros libros sobre el asunto, puede que hayas encontrado las notaciones de rango de caracteres `[A -Z]` o `[a – z]`. Son notaciones tradicionales de Unix y funcionaban en las versiones antiguas de Linux también. Pueden funcionar todavía, pero tienes que ser muy cuidadoso con ellas porque no producirán los resultados esperados a menos que estén adecuadamente configuradas. Por ahora, deberías evitar utilizarlas y usar las clases de caracteres en su lugar.

Los comodines funcionan en la GUI también

Los comodines son especialmente valiosos no sólo porque se usan tan frecuentemente en la línea de comandos, sino que también son soportados por algunos gestores gráficos de archivos.

- En **Nautilus** (el gestor de archivos de GNOME), puedes seleccionar archivos utilizando el elemento de menú Edit/Select Pattern. Sólo introduce un patrón de selección de archivos con comodines y los archivos que actualmente se vean en el directorio se marcarán para seleccionarlos.
- El algunas versiones de **Dolphin** y **Konqueror** (los gestores de archivos de KDE), puedes introducir comodines directamente en la barra de direcciones. Por ejemplo, si quieres ver

todos los archivos que empiecen por “u” minúscula en el directorio /usr/bin, introduce “/usr/bin/u*” en la barra de direcciones y mostrará el resultado.

Muchas ideas originales de la línea de comandos han pasado a la interfaz gráfica también. Ésta es una de las muchas cosas que hace al escritorio Linux tan poderoso.

mkdir – Crear directorios

El comando `mkdir` se utiliza para crear directorios. Funciona así:

```
mkdir directorio...
```

Una nota sobre la notación: Cuando tres puntos siguen a un argumento en la descripción de un comando (como el de arriba), significa que el argumento puede ser repetido, osea:

```
mkdir dir1
```

crearía un único directorio llamado “dir1”, mientras

```
mkdir dir1 dir2 dir3
```

crearía tres directorios llamados “dir1”, “dir2” y “dir3”.

cp – Copiar archivos y directorios

El comando `cp` copia archivos o directorios. Puede usarse de dos formas difrerentes:

```
cp item1 item2
```

para copiar el archivo o directorio “item1” al archivo o directorio “item2” y:

```
cp item... directorio
```

para copiar múltiples elementos (tanto archivos como directorios) a un directorio.

Opciones útiles y ejemplos

Aquí tenemos algunas de las opciones más usadas (la opción corta y la opción larga equivalente) para `cp`:

Tabla 4-4: Opciones cp

Opción	Significado
`-a, --archive`	Copia los archivos y directorios y todos sus atributos, incluyendo propietarios y permisos. Normalmente, las copias toman los atributos por defecto del usuario que realiza la copia.
`-i, --interactive`	Antes de sobrescribir un archivo existente, pide al usuario confirmación. **Si esta opción no es especificada, cp sobrescribirá silenciosamente los archivos.**
`-r, --recursive`	Copia recursivamente los directorios y su contenido. Esta opción (o la opción `-a`) se requiere cuando se copian directorios.

`-u, --update`	Cuando copiamos archivos de un directorio a otro, sólo copia archivos que o no existan, o sean más nuevos que el existente correspondiente, en el directorio de destino.
`-v, --verbose`	Muestra mensajes informativos cuando la copia se ha completado.

Tabla 4-5: Ejemplo de cp

Comando	Resultado
`cp archivo1 archivo2`	Copia el *archivo1* al *archivo2*. Si el *archivo2* existe, se sobrescribe con el contenido del *archivo1*. Si el *archivo2* no existe, se crea.
`cp -i archivo1 archivo2`	Igual que el anterior, salvo que si el *archivo2* existe, el usuario es preguntado antes de que sea sobrescrito.
`cp archivo1 archivo2 directorio1`	Copia el *archivo1* y el *archivo2* en el directorio1. El *directorio1* debe existir.
`cp directorio1/* directorio2`	Usando un comodín, todos los archivos en *directorio1* se copian a *directorio2*. d*irectorio2* debe existir.
`cp -r directorio1 directorio2`	Copia del contenido del *directorio1* al *directorio2*. Si el *directorio2* no existe, se crea y, después de la copia, contendrá los mismos elementos que el *directorio1*. Si el *directorio2* existe, entonces el *directorio1* (y su contenido) se copiará dentro de *directorio2*.

mv – Mover y renombrar archivos

El comando `mv` tanto mueve como renombra archivos, dependiendo de como se use. En ambos casos, el nombre de archivo original no existirá tras la operación. `mv` se usa de forma muy parecida a `cp`:

```
mv item1 item2
```

para mover o renombrar el archivo o directorio “item1” a “item2” o:

```
mv item... directorio
```

para mover uno o más elementos de un directorio a otro.

Opciones útiles y ejemplos

`mv` comparte muchas de las mismas opciones que `cp`:

Tabla 4-6: Opciones de mv

Opción	Significado
`-i, --interactive`	Antes de sobrescribir un archivo existente, pide confirmación al usuario. **Si esta opción no está especificada, mv sobrescribirá silenciosamente los archivos**.
`-u, --update`	Cuando movemos archivos de un directorio a otro, sólo

	mueve archivos que, o no existen, o son más nuevos que los archivos correspondientes existentes en el directorio de destino.
`-v, --verbose`	Muestra mensajes informativos cuando el movimiento ha sido realizado.

Tabla 4-7: Ejemplos de mv

Comando	Resultados
`mv` *`archivo1 archivo2`*	Mueve el *archivo1* al *archivo2*. **Si el *archivo2* existe, se sobrescribe con el contenido de *archivo1*.** Si *archivo2* no existe, se crea. **En ambos casos, *archivo1* deja de existir.**
`mv -i` *`archivo1 archivo2`*	Igual que el anterior, excepto que si el *archivo2* existe, el usuario es preguntado antes de que sea sobrescrito.
`mv` *`archivo1 archivo2 directorio1`*	Mueve *archivo1* y *archivo2* al *directorio1*.*directorio1* tiene que existir previamente.
`mv` *`directorio1 directorio2`*	Si el *directorio2* no existe, crea el *directorio2* y mueve el contenido de *directorio1* dentro de *directorio2* y borra el *directorio1*. Si *directorio2* existe, mueve *directorio1* (y su contenido) dentro del *directorio2*.

rm – Eliminar archivos y directorios

El comando `rm` se utiliza para eliminar (borrar) archivos y directorios:

```
rm elemento...
```

donde “elemento” es uno o más archivos o directorios.

Opciones útiles y ejemplos

Aquí tienes algunas de las opciones más comunes de `rm`:

Tabla 4-8: Opciones de rm

Opción	Significado
`-i, --interactive`	Antes de borrar un archivo existente, pide la confirmación del usuario. **Si esta opción no es especificada, `rm` borrará silenciosamente los archivos.**
`-r, --recursive`	Borra directorios recursivamente. Ésto significa que si un directorio que estamos borrando contiene subirectorios, los borra también. Para borrar un directorio, esta opción debe ser especificada.
`-f, --force`	Ignora archivos no existentes y no pregunta. Ésto prevalece sobre la opción `–-interactive`.
`-v, --verbose`	Muestra mensajes informativos cuando el borrado es realizado.

Tabla 4-9: Ejemplos de rm

Comando	Resultados
`rm archivo1`	Borra el *archivo1* silenciosamente
`rm -i archivo1`	Igual que antes, excepto que pide confirmación al usuario antes de borrarlo.
`rm -r archivo1 directorio1`	Borra el *archivo1* y el *directorio1* y su contenido
`rm -rf archivo1 directorio1`	Igual que antes, excepto que si *archivo1* o *directorio1* no existen, `rm` continuará silenciosamente.

¡Ten cuidado con rm!

Los sistemas operativos como-Unix, por ejemplo Linux, no tienen un comando para restaurar archivos. Una vez que borras algo con `rm`, se ha ido. Linux asume que eres listo y sabes lo que haces.

Ten especial cuidado con los comodines. Piensa en este ejemplo clásico. Digamos que quieres borrar sólo los archivos HTML en un directorio. Para hacerlo, escribes:

```
rm *.html
```

lo cual es correcto, pero si accidentalmente dejas un espacio entre el "*" y el ".html" como aquí:

```
rm * .html
```

el comando rm borrará todos los archivos en el directorio y luego se quejará de que no hay ningún archivo llamado ".html".

Aquí tienes un práctico consejo. Cada vez que uses comodines con `rm` (¡además de comprobar cuidadosamente tu escritura!), prueba el comodín primero con `ls`. Esto te permitirá ver los archivos que van a ser borrados. Entonces pulsa la tecla de la flecha hacia arriba para recuperar el comando y reemplaza el `ls` con `rm`.

ln – Crear Enlaces

El comando `ln` se usa para crear tanto enlaces duros como enlaces simbólicos. Se utiliza de una de estas dos formas:

```
ln archivo link
```

para crear un enlace duro, y:

```
ln -s elemento link
```

para crear un enlace simbólico donde "elemento" es un archivo o un directorio.

Enlaces duros

Los enlaces duros son la forma original de Unix para crear enlaces, comparados con los enlaces simbólicos, que son más modernos. Por defecto, todo archivo tiene un enlace duro que le da al

archivo su nombre. Cuando creamos un enlace duro, creamos una entrada de directorio adicional para un archivo. Los enlaces duros tienen dos limitaciones importantes:

1. Un enlace duro no puede enlazar a un archivo fuera de su propio sistema de archivos. Ésto significa que un enlace no puede enlazar a un archivo que no está en la misma partición de disco que el enlace mismo.
2. Un enlace duro no puede enlazar a un directorio.

Un enlace duro es indistinguible del fichero mismo. Al contrario que un enlace simbólico, cuando listas el contenido de un directorio que contiene un enlace duro no veras ninguna indicación especial para el enlace. Cuando un enlace duro se borra, el enlace se elimina pero el contenido del fichero continua existiendo (o sea, su espacio no es liberado) hasta que todos los enlaces al archivo son borrados.

Es importante ser conscientes de los enlaces duros porque los podrías encontrar de vez en cuando, pero las prácticas modernas prefieren los enlaces simbólicos, que veremos a continuación.

Enlaces simbólicos

Los enlaces simbólicos se crearon para solucionar las limitaciones de los enlaces duros. Los enlaces simbólicos funcionan creando un tipo especial de archivo que contiene un texto apuntando al archivo o directorio enlazado. En este aspecto, funcionan de forma muy parecida a los accesos directos de Windows, pero precedieron a esta característica de Windows en muchos años ;-)

Un archivo enlazado por un enlace simbólico, y el propio enlace simbólico son casi indistinguibles uno del otro. Por ejemplo, si escribes algo en el enlace simbólico, el archivo enlazado será modificado también. De todas formas cuando borras un enlace simbólico, sólo el enlace se borra, no el archivo. Si el archivo es borrado antes que el enlace simbólico, el enlace continuará existiendo, pero no apuntará a nada. En este caso, el enlace se dice que está *roto*. En muchas implementaciones, el comando `ls` mostrará los enlaces rotos en un color distintivo, como rojo, para revelar su presencia.

El concepto de enlace puede parecer muy confuso, pero aguanta. Vamos a probar todo ésto y, con suerte, quedará claro.

Construyamos un terreno de juego

Como vamos a realizar manipulación real de archivos, construiremos un sitio seguro para “jugar” con nuestros comandos de manipulación de archivos. Primero necesitamos un directorio donde trabajar. Crearemos uno en nuestro directorio home y lo llamaremos “playground”.

Creando directorios

El comando `mkdir` se usa para crear un directorio. Para crear nuestro terreno de juego primero nos aseguraremos de que estamos en nuestro directorio home y luego crearemos nuestro nuevo directorio:

```
[me@linuxbox ~]$ cd

[me@linuxbox ~]$ mkdir playground
```

Para hacer nuestro terreno de juego un poco más interesante, crearemos un par de directorios dentro llamados “dir1” y “dir2”. Para hacerlo, cambiaremos nuestro directorio de trabajo actual a `playground` y ejecutaremos otro `mkdir`:

```
[me@linuxbox ~]$ cd playground
```

```
[me@linuxbox playground]$ mkdir dir1 dir2
```

Fíjate que el comando `mkdir` acepta múltiples argumentos permitiéndonos crear ambos directorios con un único comando.

Copiando archivos

A continuación, pongamos algunos datos en nuestro terreno de juego. Lo haremos copiando un archivo. Utilizando el comando `cp`, copiaremos el archivo `passwd` del directorio `/etc` al directorio de trabajo actual:

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Fíjate cómo hemos usado la abreviatura para el directorio de trabajo actual, el punto que inicia la ruta. Así que ahora si ejecutamos un `ls`, veremos nuestro archivo:

```
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me me 4096 2008-01-10 16:40 dir1
drwxrwxr-x 2 me me 4096 2008-01-10 16:40 dir2
-rw-r--r-- 1 me me 1650 2008-01-10 16:07 passwd
```

Ahora, sólo por divertirnos, repitamos la copia usando la opción "-v" (verbose) para ver que hace:

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

El comando `cp` realiza la copia otra vez, pero esta vez muestra un mensaje conciso indicando que operación estaba realizando. Fíjate que `cp` sobrescribe la primera copia sin ningún aviso. De nuevo es un caso de `cp` asumiendo que sabes lo que estás haciendo. Para tener un aviso, incluiremos la opción "-i" (interactive):

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite `./passwd'?
```

Responder a la consola escribiendo un "y" hará que el archivo se sobrescriba, cualquier otro carácter (por ejemplo, "n") hará que `cp` deje al archivo en paz.

Moviendo y renombrando archivos

Bien, el nombre "passwd" no parece muy divertido y esto es un terreno de juego, así que lo cambiaremos por otra cosa:

```
[me@linuxbox playground]$ mv passwd fun
```

Divirtámonos un poco más moviendo nuestro archivo renombrado a cada directorio y volvamos atrás de nuevo:

```
[me@linuxbox playground]$ mv fun dir1
```

para moverlo primero al directorio `dir1`, luego:

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

para moverlo de `dir1` a `dir2`, después:

```
[me@linuxbox playground]$ mv dir2/fun .
```

para finalmente traerlo de nuevo al directorio de trabajo actual. A continuación, veamos el efecto de `mv` en los directorios. Primero moveremos nuestro archivo de datos al interior del `dir1` de nuevo:

```
[me@linuxbox playground]$ mv fun dir1
```

luego moveremos `dir1` dentro de `dir2` y lo confirmaremos con `ls`:

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me me 4096 2008-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me me 1650 2008-01-10 16:33 fun
```

Fíjate que como `dir2` ya existía, `mv` movió `dir1` dentro de `dir2`. Si `dir2` no hubiera existido, `mv` habría renombrado `dir1` a `dir2`. Finalmente, pondremos todo como estaba:

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

Creando enlaces duros

Ahora intentaremos crear algunos enlaces. Primero los enlaces duros. Crearemos enlaces a nuestro archivo de datos de la siguiente forma:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

Así que ahora tenemos cuatro instancias del archivo “fun”. Echemos un vistazo a nuestro directorio playground:

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me me 4096 2008-01-14 16:17 dir1
drwxrwxr-x 2 me me 4096 2008-01-14 16:17 dir2
-rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun
-rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun-hard
```

Una cosa que notarás es que el segundo campo en el listado para `fun` y `fun-hard` contiene en ambos un “4” que es el número de enlaces duros que existen ahora para el archivo. Recordarás que un archivo tiene siempre al menos un enlace duro porque el nombre del archivo se crea con un enlace. Entonces, ¿cómo sabemos que `fun` y `fun-hard` son, en realidad, el mismo archivo? En este caso, `ls` no es muy útil. Aunque podemos ver que `fun` y `fun-hard` tienen los dos el mismo tamaño (campo 5), nuestro listado no nos ofrece una forma de estar seguro. Para solucionar el problema, vamos a tener que cavar un poco más hondo.

Cuando pensamos en los enlaces duros, puede ayudarnos imaginar que los archivos está compuestos de dos partes: la parte de datos que alberga el contenido del archivo y la parte del nombre que contiene el nombre del archivo. Cuando creamos enlaces duros, realmente estamos creando partes

de nombre adicionales que se refieren todos a la misma parte de datos. El sistema asigna una cadena de sectores del disco llamada inodo, la cual es asociada con la parte del nombre. Cada enlace duro por lo tanto se refiere a un inodo específico que alberga el contenido del archivo.

El comando `ls` tiene una forma de revelar esta información. Se invoca con la opción “-i”:

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me me 4096 2008-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me me 4096 2008-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun
12353538 -rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun-hard
```

En esta versión de la lista, el primer campo es el número del inodo y, como podemos ver, tanto `fun` como `fun-hard` comparten el mismo número de inodo, lo que confirma que son el mismo archivo.

Creando enlaces simbólicos

Los enlaces simbólicos fueron creados para superar las dos desventajas de los enlaces duros: Los enlaces duros no pueden salir de su dispositivo físico y tampoco pueden enlazar directorios, sólo archivos. Los enlaces simbólicos son un tipo especial de archivo que contiene un texto apuntando al archivo o directorio de destino.

Crear enlaces simbólicos es similar a crear enlaces duros:

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

El primer ejemplo es muy sencillo, simplemente añadimos la opción “-s” para crear un enlace simbólico en lugar de un enlace duro. Pero ¿qué pasa con los dos siguientes? Recuerda, cuando creamos un enlace simbólico, estamos creando un texto describiendo donde está el archivo de destino con respecto al enlace simbólico. Es más fácil verlo si miramos el resultado de `ls`:

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun-hard
lrwxrwxrwx 1 me me 6 2008-01-15 15:17 fun-sym -> ../fun
```

El listado para `fun-sym` en `dir1` muestra que es un enlace simbólico con una “l” en el primer carácter del primer campo y que apunta a “../fun”, lo cual es correcto. En cuanto a la localización de `fun-sym`, `fun` está en el directorio superior. Ten en cuenta que, la longitud del archivo enlace simbólico es 6, el número de caracteres de la cadena “../fun” en lugar de la longitud del archivo al que está apuntando.

Cuando creamos enlaces simbólicos, puedes usar rutas absolutas:

```
ln -s /home/me/playground/fun dir1/fun-sym
```

o rutas relativas, como hicimos en el anterior ejemplo. Usar rutas relativas es más recomendable porque permite que un directorio que contiene enlaces simbólicos pueda ser renombrado y/o movido sin romper los enlaces.

Además de archivos normales, los enlaces simbólicos también pueden enlazar directorios:

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me me 4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir2
-rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun
-rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun-hard
lrwxrwxrwx 1 me me 3 2008-01-15 15:15 fun-sym -> fun
```

Eliminando archivos y directorios

Como vimos anteriormente, el comando `rm` se usa para borrar archivos y directorios. Vamos a usarlo para limpiar nuestro terreno de juego un poco. Primero, borraremos uno de nuestros enlaces duros:

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me me 4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir2
-rw-r--r-- 3 me me 1650 2008-01-10 16:33 fun
lrwxrwxrwx 1 me me 3 2008-01-15 15:15 fun-sym -> fun
```

Ha funcionado como esperábamos. El archivo `fun-hard` se ha ido y el contador de enlaces de `fun` muestra que se ha reducido de cuatro a tres, como indica el segundo campo del listado del directorio. A continuación, borraremos el archivo `fun`, y sólo por divertirnos, incluiremos la opción "-i" para ver qué hace:

```
[me@linuxbox playground]$ rm -i fun
rm: remove regular file `fun'?
```

Escribe "y" en la consola y el archivo se borra. Pero miremos al mensaje de `ls` ahora. ¿Ves lo que le ha pasado a `fun-sym`? Como era un enlace simbólico apuntando a un archivo que ahora no existe, el enlace está *roto*:

```
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me me 4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir2
lrwxrwxrwx 1 me me 3 2008-01-15 15:15 fun-sym -> fun
```

La mayoría de las distribuciones Linux configuran `ls` para mostrar los enlaces rotos. En Fedora, los enlaces rotos ¡se muestran en texto rojo parpadeante! La presencia de un enlace roto no es, por sí misma peligrosa pero si indica un poco de desorden. Si intentamos usar un enlace roto veremos ésto:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

Limpiemos un poco. Borraremos los enlaces simbólicos:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir1
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir2
```

Una cosa que hay que recordar sobre los enlaces simbólicos es que la mayoría de las operaciones con archivos son realizadas en el archivo de destino, no en el propio enlace. `rm` es una excepción. Cuando borras un enlace, es el enlace el que es eliminado, no el archivo de destino.

Finalmente, borraremos nuestro terreno de juego. Para hacerlo, volveremos a nuestro directorio home y usaremos rm con la opción recursiva (-r) para borrar playground y todo su contenido, incluidos los subdirectorios:

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

Creando enlaces simbólicos con la GUI

Los gestores de archivos tanto en GNOME como en KDE proporcionan un método fácil y automático para crear enlaces simbólicos. Con GNOME, presionando las teclas Ctrl-Shift mientras arrastramos un archivo creamos un enlace en lugar de copiar (o mover) el archivo. En KDE un pequeño menú aparece donde arrastremos el archivo, ofreciendo la opción de copiarlo, moverlo o enlazarlo.

Resumiendo

Hemos abarcado mucho terreno y no llevará algo de tiempo asentarlo complentamente. Realiza el ejercicio del terreno de juego una y otra vez hasta que tenga sentido. Es importante tener una buena comprensión de la manipulación básica de comandos y comodines. Siéntete libre para ampliar el ejercicio del terreno de juego añadiendo más archivos y directorios, usando comodines para archivos concretos para varias operaciones. El concepto de enlace es un poco confuso al principio, pero tómate tiempo para aprender como funcionan. Pueden ser un verdadero salvavidas.

Para saber más

- Un artículo sobre los enlaces simbólicos: http://en.wikipedia.org/wiki/Symbolic_link

Trabajando con comandos

Llegados a este punto, hemos visto una serie de misteriosos comandos, cada uno con sus misteriosas opciones y argumentos. En este capítulo, intentaremos resolver algunos de estos misterios e incluso crear algunos comandos propios. Los comandos presentados en este capítulo son:

- `type` – Indica como se interpreta el nombre de un comando
- `which` – Muestra que programa ejecutable será ejecutado
- `help` – Ofrece ayuda para funciones del shell
- `man` – Muestra el manual de un comando

- `apropos` – Muestra una lista de comandos apropiados
- `info` – Muestra información sobre un comando
- `whatis` – Muestra una descripción muy breve de un comando
- `alias` – Crea un alias para un comando

¿Qué son exactamente los comandos?

Un comando puede ser una de estas cuatro cosas:

1. **Un programa ejecutable** como todos esos archivos que vimos en `/usr/bin`. Dentro de esta categoría, los programas pueden ser *binarios compilados* como programas escritos en C y C++, o programas escritos en *lenguajes de script* como el shell, perl, python, ruby, etc.
2. **Un comando contenido en el propio shell**. `bash` soporta gran número de comandos internos llamados *shell builtins*. El comando `cd`, por ejemplo, es un shell builtin.
3. **Una función del shell**. Éstos son pequeños scripts de shell incorporados en el *entorno*. Veremos cómo configurar el entorno y cómo escribir funciones del shell en próximos capítulos, pero por ahora, sólo ten en cuenta que existen.
4. **Un alias**. Comando que podemos definir nosotros mismos, construidos a partir de otros comandos.

Identificando comandos

A menudo es útil saber exactamente cual de los cuatro tipos de comandos estamos usando y Linux proporciona un par de formas de averiguarlo.

type – Muestra de que tipo es un comando

El comando `type` es una función del shell que muestra el tipo de comando que el shell ejecutará, indicándole el nombre de un comando en particular. Funciona así:

type *comando*

donde *comando* es el nombre del comando que quieres examinar. Aquí tienes algunos ejemplos:

```
[me@linuxbox ~]$ type type
type is a shell builtin
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
[me@linuxbox ~]$ type cp
cp is /bin/cp
```

Aquí vemos los resultados para tres comandos diferentes. Fíjate en el de `ls` (realizado sobre un sistema Fedora) y en cómo el comando `ls` es realmente un alias del comando `ls` con la opción “--color=tty” añadida. ¡Ahora sabemos porqué la salida de `ls` se muestra en color!

which – Muestra la localización de un ejecutable

A veces hay más de una versión de un programa ejecutable instalada en un sistema. Aunque no es muy usual en los sistemas de escritorio, sí es más frecuente en grandes servidores. Para determinar la localización exacta de un ejecutable dado tenemos el comando `which`:

```
[me@linuxbox ~]$ which ls
/bin/ls
```

`which` sólo funciona con programas ejecutables, no con builtins ni con alias que son sustitutos de los verdaderos programas ejecutables. Cuando intentamos usar `which` en un shell builtin, por ejemplo, `cd`, no tendremos respuesta o nos dará un mensaje de error:

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in (/opt/jre1.6.0_03/bin:/usr/lib/qt-
3.3/bin:/usr/kerberos/bin:/opt/jre1.6.0_03/bin:/usr/lib/cca
che:/usr/l
ocal/bin:/usr/bin:/bin:/home/me/bin)
```

que es una forma algo complicada de decir “comando no encontrado”.

Obteniendo la documentación de un comando

Ahora que sabemos qué es un comando, podemos buscar la documentación disponible para cada tipo de comando.

help – Ofrece ayuda sobre los shell builtins

bash tiene una ayuda integrada disponible para cada shell builtin. Para usarla, escribimos “help” seguido del nombre del shell builtin. Por ejemplo:

```
[me@linuxbox ~]$ help cd
cd: cd [-L|[-P [-e]]] [dir]
Change the shell working directory.

Change the current directory to DIR. The default DIR is the
value of the HOME shell variable.

The variable CDPATH defines the search path for the
directory containing DIR. Alternative directory names in
CDPATH are separated by a colon (:). A null directory name
is the same as the current directory. If DIR begins with a
slash (/), then CDPATH is not used.

If the directory is not found, and the shell option
`cdable_vars' is set, the word is assumed to be a variable
name. If that variable has a value, its value is used for
DIR.

Options:
-L   force symbolic links to be followed
-P   use the physical directory structure without following
symbolic links
-e   if the -P option is supplied, and the current working
directory cannot be determined successfully, exit with a
non-zero status

The default is to follow symbolic links, as if `-L' were
specified.

Exit Status:
Returns 0 if the directory is changed, and if $PWD is
```

```
set successfully when -P is used; non-zero otherwise.
```

Una nota sobre la notación: Cuando aparecen corchetes en la descripción de la sintaxis de un comando, indican opciones adicionales. Una barra vertical indica que son elementos excluyentes entre ellos. En el caso del comando cd que vimos antes:

```
cd [-L|[-P[-e]]] [dir]
```

Esta notación dice que el comando `cd` podría ir seguido opcionalmente por una "-L" o una "-P" y después, si la opción "-P" está especificada la opción "-e" podría también ser incluida seguida por el argumento opcional "dir".

Mientras que la salida de `help` para el comando `cd` es concisa y precisa, esto no significa que sea un tutorial como podemos ver, también parece mencionar muchas cosas de las que aún no hemos hablado. No te preocupes. Llegaremos a ello.

--help – Muestra información de uso

Muchos programas ejecutables soportan una opción "--help" que muestra una descripción de la sintaxis y las opciones soportadas por el comando. Por ejemplo:

```
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

-Z, --context=CONTEXT (SELinux) set security context to
CONTEXT Mandatory arguments to long options are mandatory
for short options too.
-m, --mode=MODE   set file mode (as in chmod), not a=rwx –
umask
-p, --parents   no error if existing, make parent
directories as needed

-v, --verbose   print a message for each created directory
--help   display this help and exit
--version   output version information and exit

Report bugs to <bug-coreutils@gnu.org>.
```

Algunos programas no soportan la opción "--help", pero pruébalo de todas formas. A menudo el resultado es un error que revelará la misma información de uso.

man – Muestra el manual de un programa

La mayoría de los programas ejecutables pensados para usarse en la linea de comandos proveen un documento llamado *manual page* o *man page*. Un programa de paginación especial llamado `man` se usa para verlos. Se usa así:

man programa

donde "programa" es el nombre del comando que queremos ver.

Las man pages varían algo en formato pero generalmente contienen un título, una sinopsis de la sintaxis del comando, una descripción del propósito del comando, y una lista y descripción de cada

opción del comando. Las man pages, de todas formas, no suelen incluir ejemplos, y están pensadas como una referencia, no como un tutorial. Como ejemplo, probaremos a ver la man page para el comando `ls`:

```
[me@linuxbox ~]$ man ls
```

En la mayoría de los sistemas Linux, `man` usa `less` para mostrar la man page, así que todos los comandos de `less` que nos son familiares funcionan cuando está mostrado la página.

El "manual" que `man` muestra está dividido en secciones y no sólo cubre los comandos de usuario sino también los comandos del administrador de sistema, interfaces de programación, formatos de archivo y más. La tabla siguiente describe la estructura del manual:

Tabla 5-1: Organización de la man page

Sección	Contenido
1	Comandos de usuario
2	Interfaces de programación para llamadas del kernel del sistema
3	Interfaces de programación para la librería C
4	Archivos especiales como nodos de dispositivos y controladores
5	Formatos de archivo
6	Juegos y pasatiempos como protectores de pantalla
7	Miscelánea
8	Comandos de administración del sistema

A veces necesitamos buscar en una sección específica del manual para encontrar lo que estamos buscando. Esto es particularmente cierto si estamos buscando un formato de archivo que también es el nombre de un comando. Sin especificar un número de sección, siempre obtendremos el primer resultado, probablemente en la sección 1. Para especificar un número de sección, usamos man así:

```
man sección término_buscado
```

Por ejemplo:

```
[me@linuxbox ~]$ man 5 passwd
```

Ésto mostrará la man page describiendo el formato de archivo del archivo `/etc/passwd`.

apropos – Muestra comandos apropiados

También es posible buscar una lista de man pages para posibles resultados basados en un término de búsqueda. Es muy tosco pero a veces ayuda. Aquí tienes un ejemplo de una búsqueda de man pages utilizando el término de búsqueda "floppy":

```
[me@linuxbox ~]$ apropos floppy
create_floppy_devices (8) - udev callout to create all
possible floppy device based on the CMOS type
fdformat (8) - Low-level formats a floppy disk
floppy (8) - format floppy disks
gfloppy (1) - a simple floppy formatter for the GNOME
mbadblocks (1) - tests a floppy disk, and marks the
```

```
bad blocks in the FAT
mformat (1) - add an MSDOS filesystem to a low-
level formatted floppy disk
```

El primer campo de cada línea es el nombre de la man page, el segundo campo muestra la sección. Fíjate que el comando man con la opción "-k" ofrece exactamente la misma función que `apropos`.

whatis – Muestra una descripción muy breve de un comando

El programa `whatis` muestra el nombre y una descripción de una línea de una man page coincidente con una palabra especificada:

```
[me@linuxbox ~]$ whatis lsls (1) - list directory contents
```

La man page más brutal de todas

Como hemos visto, las man pages proporcionadas por Linux y otros sistemas como-Unix están pensadas como una documentación de referencia y no como tutoriales. Muchas man pages son duras de leer, pero pienso que el gran premio a la dificultad tiene que ir a la man page de `bash`. Cuando estaba haciendo mi investigación para este libro, revisé muy cuidadosamente que estaba cubriendo la mayoría de los temas. Cuando la imprimí, eran alrededor de 80 páginas extremadamente densas, y su estructura no tenían absolutamente ningún sentido para un usuario nuevo.

Por otra parte, es muy precisa y concisa, además de ser extremadamente completa. Así que échale un vistazo si te atreves y imagínate el día en que puedas leerla y tenga sentido.

info – Muestra un archivo de información de un programa

El Proyecto GNU proporciona una alternativa a las man pages para sus programas, llamada "info." Las páginas info se muestran con un programa lector llamado, muy apropiadamente, `info`. Las páginas info están *hyperenlazadas* como las páginas web. Aquí tenemos un ejemplo:

```
File: coreutils.info, Node: ls invocation, Next: dir
invocation,
Up: Directory listing



10.1 `ls': List directory contents
==================================



The `ls' program lists information about files (of any
type, including directories). Options and file arguments
can be intermixed arbitrarily, as usual.



For non-option command-line arguments that are directories,
by default `ls' lists the contents of directories, not
recursively, and omitting files with names beginning with
`.'. For other non-option arguments, by default `ls' lists
just the filename. If no non-option argument is specified,
```

```
`ls' operates on the current directory, acting as if it had
been invoked with a single argument of `.'.


By default, the output is sorted alphabetically, according
to the --zz-Info: (coreutils.info.gz)ls invocation, 63
lines –Top----------
```

El programa `info` lee *archivos info*, que tienen una estructura de árbol con *nodos* individuales, cada uno de los cuales contiene un asunto concreto. Los archivos info contienen hipervínculos que permiten movernos de un nodo a otro. Un hipervínculo puede identificarse por su asterisco delante, y se activa colocando el cursor encima y presionando la tecla enter.

Para invocar a `info`, escribe "info" seguido opcionalmente del nombre de un programa. A continuación tienes una tabla de comandos usados para controlar el lector mientras muestra una página info:

Tabla 5-2: Comandos info

Comando	Acción
`?`	Muestra la ayuda de un comando
`PgUp` o `Borrar`	Muestra la página previa
`PgDn` o `Espacio`	Muestra la siguiente página
`n`	Siguiente – Muestra el siguiente nodo
`p`	Previo – Muestra el nodo anterior
`u`	Up – Muestra el nodo padre del nodo mostrado actualmente, normalmente un menú.
`Enter`	Sigue el hipervínculo que está bajo el cursor
`q`	Salir

La mayoría de los programas de la línea de comandos que hemos visto hasta ahora son parte del paquete "coreutils" del Proyecto GNU, así que escribiendo:

```
[me@linuxbox ~]$ info coreutils
```

mostrará una página con un menú con hipervínculos a cada programa contenido en el paquete coreutils.

README y otros archivos con documentación de programas

Muchos paquetes de software instalados en tu sistema tienen archivos de documentación localizados en el directorio `/usr/share/doc`. La mayoría de ellos están en formato de texto plano y puede ser leidos con `less`. Algunos de estos archivos están en formato HTML y pueden ser abiertos con un explorador web. Encontraremos algunos archivos que terminan con la extensión ".gz". Ésto indica que han sido comprimidos con el programa de compresión `gzip`. El paquete gzip incluye una versión especial de `less` llamada `zless` que mostrará el contenido de los archivos de texto comprimidos con gzip.

Creando tus propios comandos con alias

Ahora ¡para nuestra primera experiencia con la programación! crearemos un comando propio usando el comando `alias`. Pero antes de empezar, necesitamos desvelar un pequeño truco de la línea de comandos. Es posible poner más de un comando en una línea separando cada comando con un punto y coma. Funciona así:

comando1; comando2; comando3...

Aquí tenemos el ejemplo que usaremos:

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin games kerberos lib64 local share tmp
etc include lib libexec sbin src
/home/me
[me@linuxbox ~]$
```

Como podemos ver, hemos combinado tres comandos en una línea. Primero hemos cambiado el directorio a `/usr` luego hemos listado el directorio y finalmente hemos vuelto al directorio original (usando '`cd -`') con lo que hemos terminado donde empezamos. Ahora convirtamos esta secuencia en un nuevo comando usando `alias`. La primera cosa que tenemos que hacer es inventarnos un nombre para nuestro nuevo comando. Probemos “test”. Antes de hacerlo, sería una buena idea averiguar si el nombre “test” ya está siendo usado. Para averiguarlo, podemos usar el comando `type` de nuevo:

```
[me@linuxbox ~]$ type test
test is a shell builtin
```

Ups! El nombre “test” ya está cogido. Probemos “foo”:

```
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

¡Genial! “foo” no está cogido. Así que creemos nuestro alias:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

Fíjate en la estructura de este comando:

alias name='string'

Tras el comando “alias” le damos a alias un nombre seguido inmediatamente (si espacio en blanco) por un signo igual, seguido inmediatamente por una cadena entre comillas simples conteniendo los comandos que se asignarán al nombre. Después de definir nuestro alias, puede ser usado donde quiera que el shell acepte comandos. Probemos:

```
[me@linuxbox ~]$ foo
bin games kerberos lib64 local share tmp
etc include lib libexec sbin src
/home/me
[me@linuxbox ~]$
```

También podemos usar el comando `type` de nuevo para ver nuestro alias:

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls ; cd -'
```

Para eliminar un alias, se usa el comando `unalias`, así:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

Aunque evitamos a propósito nombrar nuestro alias con un nombre de comando existente, no es raro que se haga. A menudo se hace para aplicar una opción que se utilice a menudo en cada invocación de un comando común. Por ejemplo, vimos antes que el comando `ls` a menudo es un alias con soporte para colores:

```
[me@linuxbox ~]$ type ls
ls is aliased to `ls –color=tty'
```

Para ver todos los alias definidos en nuestro entorno, utiliza el comando `alias` sin argumentos. Aquí tienes algunos de los alias definidos por defecto en un sistema Fedora. Pruébalo e imagina para que sirve cada uno:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls –color=tty'
```

Hay un pequeño problema con la definición de alias en la linea de comandos. Se esfuman cuando la sesión de shell se cierra. En próximos capítulos, veremos como añadir nuestros alias a archivos que establecen el entorno cada vez que nos logueamos, pero por ahora, disfruta del hecho de que hemos dado nuestro primer paso, aunque sea pequeño, ¡en el mundo de la programación en shell!

Resumiendo

Ahora que hemos aprendido como encontrar documentación sobre los comandos, échale un vistazo a la documentación de todos los comandos que hemos visto hasta ahora. Estudia que opciones adicionales están disponibles y ¡prúebalas!

Para saber más

Hay muchas fuentes de documentación online para Linux y la línea de comandos. Aquí tienes algunas de las mejores:

- El *Manual de Referencia de Bash* es una guía de referencia del shell `bash`. Sigue siendo sólo un trabajo de referencia pero contiene ejemplos y es más fácil de leer que la man page de `bash`.

http://www.gnu.org/software/bash/manual/bashref.html

- El `Bash FAQ` contiene respuestas a preguntas frecuentes sobre bash. Esta lista está dirigida usuarios de intermedios a avanzados, pero contiene un montón de buena información.

http://mywiki.wooledge.org/BashFAQ

- El Proyecto GNU provee una extensa documentación para sus programas, los cuales forman el corazón de la experiencia con la linea de comandos de Linux. Puedes ver una lista completa aquí:

http://www.gnu.org/manual/manual.html

- Wikipedia tiene un interesante artículo sobre las man pages:

http://en.wikipedia.org/wiki/Man_page

Redirección

En esta lección vamos a desvelar la que podría ser la mejor característica de la línea de comandos. Se llama *redirección I/O*. I/O significa *input/output* y con esta aplicación puedes redirigir la salida y entrada de los comandos hacia o desde archivos, así como conectar múltiples comandos juntos en poderosos "*conductos*" o *pipelines* de comandos. Para mostrar esta aplicación introduciremos los siguientes comandos:

- `cat` – Concatena archivos
- `sort` – Ordena líneas de texto
- `uniq` – Reporta u omite lineas repetidas
- `grep` – Imprime lineas que coincidan con un patrón
- `wc` – Imprime el número de líneas, palabras y bytes para cada archivo
- `head` – Imprime la primera parte de un archivo
- `tail` – Imprime la última parte de un archivo
- `tee` – Lee de la entrada estándar y escribe en la salida estándar y en archivos

Entrada, salida y error estándar

Muchos de los programas que hemos usado hasta ahora producen algún tipo de salida. Esta salida, a menudo es de dos tipos. Primero, tenemos los resultados del programa; o sea, los datos que el programa está diseñado para producir, y segundo, tenemos mensajes de estado y de error que nos dice como va el programa. Si miramos un comando como `ls`, podemos ver que muestra sus resultados y sus mensajes de error en la pantalla.

Siguiendo con el tema Unix de "todo es un archivo", programas como `ls` en realidad mandan sus resultados a un archivo especial llamado *standard output* o salida estándar (a menudo expresado como stdout) y sus mensajes de estado a otro archivo llamado *standard error* o error estándar (*stderr*). Por defecto, tanto la salida estándar como el error estándar están enlazados a la pantalla y no se guardan en un archivo en el disco.

Además muchos programas toman la entrada de una aplicación llamada *standard input* o entrada estándar (*stdin*) que está, por defecto, asociada al teclado.

La redirección I/O nos permite cambiar donde va la salida y de donde viene la entrada. Normalmente, la salida va a la pantalla y la entrada viene del teclado, pero con la redirección I/O, podemos cambiarlo.

Redirigiendo la salida estándar

La redirección I/O nos permite redefinir donde va la salida estándar. Para redirigir la salida estándar a otro archivo en lugar de a la pantalla, usamos el operador de redirección ">" seguido del nombre del archivo. ¿Para qué querríamos hacer ésto? A menudo es útil almacenar la salida de un comando en un archivo. Por ejemplo, podríamos decirle al shell que mande la salida del comando ls al archivo `ls-output.txt` en lugar de a la pantalla:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Aquí, hemos creado un largo listado del directorio `/usr/bin` y hemos mandado los resultados al archivo `ls-output.txt`. Examinemos la salida redirigida del comando:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me me 167878 2008-02-01 15:07 ls-output.txt
```

Bien, un bonito y largo archivo de texto. Si vemos el archivo con `less`, veremos que el archivo `ls-output.txt` en efecto contiene el resultado del nuestro comando ls:

```
[me@linuxbox ~]$ less ls-output.txt
```

Ahora, repitamos nuestra prueba de redirección, pero esta vez con un giro. Cambiaremos el nombre del directorio a uno que no exista:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

Recibimos un mensaje de error. Ésto tiene sentido ya que especificamos el directorio `/bin/usr` que no existe, pero ¿por qué ha mostrado el mensaje de error en la pantalla en lugar que ser redirigido al archivo `ls-output.txt`? La respuesta es que el programa ls no manda sus mensajes de error a la salida estándar. En lugar de eso, como la mayoría de los programas bien escritos de Unix, manda sus mensajes de error al error estándar. Como sólo hemos redirigido la salida estándar y no el error estándar, el mensaje de error todavía es enviado a la pantalla. Veremos como redirigir el error estándar en un minuto, pero primero, veamos que ha ocurrido a nuestro archivo de salida:

```
[me@linuxbox ~]$ ls -l ls-output.txt-rw-rw-r-- 1 me me 0
2008-02-01 15:08 ls-output.txt
```

¡El archivo ahora tiene tamaño cero! Ésto es porque, cuando redirigimos la salida con el operador de redirección “>”, el archivo de destino siempre se sobreescribe desde el principio. Como nuestro comando `ls` no generó resultados y sólo un mensaje de error, la operación de redirección comenzó a reescribir el archivo y paró a causa del error, truncándose. De hecho, si alguna vez necesitamos realmente un archivo truncado (o crear un archivo nuevo vacío) podemos usar un truco como éste:

```
[me@linuxbox ~]$ > ls-output.txt
```

Simplemente usando el operador de redirección con ningún comando precediéndolo truncará un archivo existente o creará un archivo nuevo vacío.

Pero, ¿cómo podemos añadir la salida redirigida a un archivo en lugar de sobreescribir el archivo desde el principio? Para eso, usamos el operador de redirección “>>”, así:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

Usando el operador “>>” tendremos como resultado que la salida se añadirá al archivo. Si el archivo no existe, se creará igual que como con el operador “>” que hemos estado usando. Probémoslo:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me me 503634 2008-02-01 15:45 ls-output.txt
```

Hemos repetido el comando tres veces resultando un archivo de salida tres veces más grande.

Redirigiendo el error estándar

Redirigir el error estándar carece de la facilidad de un operador de redirección dedicado. Para redirigir el error estándar debemos referirnos a su *descriptor de archivo (file descriptor)*. Un programa puede producir salidas en una de varias cadenas de archivos numeradas. Aunque nos hemos referido a las tres primeras de estas cadenas de archivos como entrada estándar, salida estándar y error estándar, el shell se refiere a ellas internamente como los descriptores de archivo 0, 1 y 2 respectivamente. El shell proporciona una notación para los archivos redirigidos utilizando el número descriptor de archivo. Como el error estándar es el mismo que el descriptor de archivo número 2, podemos redirigir el error estándar con esta notación:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

El descriptor de archivo "2" se coloca inmediatamente antes del operador de redirección para realizar la redirección del error estándar al archivo `ls-error.txt`.

Redirigiendo la salida estándar y el error estándar a un archivo

Hay casos en que querremos capturar toda la salida de un comando a un archivo. Para hacerlo, debemos redirigir tanto la salida estándar como el error estándar al mismo tiempo. Hay dos formas de hacerlo. Primero, la forma tradicional, que funciona con versiones antiguas del shell:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Utilizando este método, conseguiremos dos redirecciones. Primero redirigimos la salida estándar al archivo `ls-output.txt` y después redirigimos el descriptor de archivo 2 (error estándar) al descriptor de archivo 1 (salida estándar) usando la notación 2>&1.

Fíjate que el orden de las redirecciones es fundamental. La redirección del error estándar siempre debe ocurrir despúes de redirigir la salida estándar o no funcionará. En el ejemplo anterior,

```
>ls-output.txt 2>&1
```

se redirige el error estándar al archivo `ls-output.txt`, pero si cambiamos el orden a

```
2>&1 >ls-output.txt
```

el error estándar es redirigido a la pantalla.

Versiones recientes de `bash` proporcionan un segundo, y más eficiente método para realizar esta redirección combinada:

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

En este ejemplo, usamos la notación simple &> para redirigir tanto la salida estándar como el error estándar al archivo `ls-output.txt`. Podrías también añadir la salida estándar y el error estándar a un archivo existente así:

```
[me@linuxbox ~]$ ls -l /bin/usr &>> ls-output.txt
```

Eliminando salidas innecesarias

Algunas veces "el silencio es oro", y no queremos salida de un comando, sólo queremos ignorarla.

Ésto se aplica particularmente a los mensajes de estado y de error. El sistema proporciona una forma de hacerlo redireccionando la salida a un archivo especial llamado “dev/null”. Este archivo es un dispositivo de sistema llamado un *cubo de bits* que acepta entradas y no hace nada con ellas. Para suprimir los mensajes de error de un comando, hacemos esto:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

Redireccionando la entrada estándar

Hasta ahora, no hemos encontrado ningún comando que haga uso de la entrada estándar (en realidad sí lo hemos hecho, pero revelaremos la sorpresa un poco más tarde), así que necesitamos presentar uno.

cat – Concatenando archivos

El comando `cat` lee uno o más archivos y los copia a la salida estándar de la siguiente forma:

```
cat [file...]
```

En la mayoría de los casos, puedes pensar que `cat` es un análogo al comando `TYPE` de DOS. Puedes usarlo para mostrar archivos sin paginar, por ejemplo:

```
[me@linuxbox ~]$ cat ls-output.txt
```

mostrará el contenido del archivo `ls-output.txt`. `cat` a menudo se usa para mostrar archivos de texto cortos. Como cat puede aceptar más de un archivo como argumento, puede ser usado también para unir archivos. Imagina que hemos descargado un gran archivo que ha sido dividido en múltiples partes (los archivos multimedia a menudo está divididos de esta forma en Usenet), y queremos unirlos de nuevo. Si los archivos se llamaran:

```
movie.mpeg.001 movie.mpeg.002 ... movie.mpeg.099
```

podríamos unirlos de nuevo con este comando:

```
cat movie.mpeg.0* > movie.mpeg
```

Como los comodines siempre expanden sus resultados en orden alfabético, los argumentos se distribuirán en el orden correcto.

Todo esto está muy bien, pero ¿qué tienen esto que ver con la entrada estándar? Nada todavía, pero probemos algo más. Que pasa si usamos “cat” sin argumentos:

```
[me@linuxbox ~]$ cat
```

No ocurre nada, sólo se queda quieto como si se hubiera quedado colgado. Podría parecer eso, pero realmente está haciendo lo que se supone que debe hacer.

Si no le damos argumentos a cat, lee de la entrada estándar y como la entrada estándar está, por defecto, asignada al teclado, ¡está esperando a que tecleemos algo! Prueba a añadir el siguiente texto y pulsa Enter:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
```

Ahora, pulsa `Ctrl-d` (p.ej., pulsa la tecla Ctrl y luego pulsa “d”) para decirle a `cat` que ha alcanzado el *final del archivo* (EOF – end of file) en el salida estándar:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
The quick brown fox jumped over the lazy dog.
```

En ausencia de nombres de archivo como argumentos, `cat` copia la entra estándar a la salida estándar, así que vemos nuestra línea de texto repetida. Podemos usar este comportamiento para crear archivos de texto cortos. Digamos que queremos crear un archivo llamado “lazy_dog.txt” conteniendo el texto de nuestro ejemplo. Podríamos hacer ésto:

```
[me@linuxbox ~]$ cat > lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Escribe el comando seguido del texto que queremos colocar en el archivo. Recuerda pulsar `Ctrl-d` al final. Usando la línea de comandos, ¡hemos implementado el procesador de texto más tonto del mundo! Para ver nuestros resultados, podemos usar cat para copiar el archivo a stdout de nuevo:

```
[me@linuxbox ~]$ cat lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Ahora que conocemos como `cat` acepta la entrada estándar, además de nombres de archivo como argumentos, probemos redirigiendo la entrada estándar:

```
[me@linuxbox ~]$ cat < lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Usando el operador de redirección “<”, hemos cambiado el origen de la entrada estándar del teclado al archivo `lazy_dog.txt`. Vemos que el resultado es el mismo que pasarle un nombre de archivo como argumento, pero sirve para demostrar el uso de un archivo como fuente de la entrada estándar. Otros comandos hacer un mejor uso de la entrada estándar, como veremos pronto.

Antes de seguir, echa un vistazo a la man page de `cat`, ya que tiene varias opciones interesantes.

Pipelines (Tuberías)

La capacidad de los comandos de leer datos de la entrada estándar y mandarlos a la salida estándar la utiliza una función del shell llamada pipeline (tubería). Usando el operador pipe (tubo) “|” (la barra vertical), la salida estándar de un comando puede ser canalizada hacia la entrada estándar de otro:

```
comando1 | comando2
```

Para demostrarlo plenamente, vamos a necesitar algunos comandos. ¿Recuerdas que dijimos que había uno que ya conocíamos que acepta entrada estándar? Es `less`. Podemos usar `less` para mostrar, página a página, la salida de cualquier comando que mande sus resultados a la salida estándar:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

¡Ésto es tremendamente práctico! Usando esta técnica, podemos examinar convenientemente la salida de cualquier comando que produzca salida estándar.

La diferencia entre > y |

A primera vista, podría ser difícil de comprender la redirección realizada por el operador pipeline | frente a la del operador de redirección >. Simplificando, el operador de redirección conecta un

comando con un archivo mientras que el operador pipeline conecta la salida de un comando con la entrada de un segundo comando.

comando1 > archivo1
comando1 | comando2

Mucha gente intentará lo siguiente cuando están aprendiendo los pipelines, "sólo mira lo que sucede."

comando1 > comando2

Respuesta: A veces algo realmente malo.

Aquí tenemos un ejemplo real enviado por un lector que estaba administrando una aplicación de servidor basada el Linux. Como superusuario, hizo ésto:

```
# cd /usr/bin
# ls > less
```

El primer comando le colocó en el directorio donde están almacenados la mayoría de los programas y el segundo comando le dijo al shell que sobrescriba el archivo `less` con la salida del comando `ls`. Como el directorio `/usr/bin` ya contenía un archivo llamado "less" (el programa `less`), el segundo comando sobrescribió el archivo del programa `less` con el texto de `ls` y en consecuencia destruyendo el programa `less` en su sistema.

La lección aquí es que el operador de redirección crea o sobrescribe archivos silenciosamente, así que necesitas tratarlo con mucho respeto.

Filtros

Los pipelines a menudo se usan para realizar complejas operaciones con datos. Es posible poner varios comandos juntos dentro de un pipeline. Frecuentemente, a los comandos usados de esta forma se les llama *filtros*. Los filtros toman entradas, la cambian en algo y las mandan a la salida. El primero que probaremos es `sort`. Imagina que queremos hacer una lista combinada de todos los programas ejecutables en `/bin` y en `/usr/bin`, que los ponga en orden y los veamos:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

Como hemos especificado dos directorios (`/bin` y `/usr/bin`), la salida de `ls` debería haber consistido en dos listas ordenadas, una para cada directorio. Pero incluyendo `sort`, en nuestro pipeline, hemos cambiado los datos para producir una única lista ordenada.

uniq – Muestra u omite líneas repetidas

El comando `uniq` a menudo se usa junto con `sort`. `uniq` acepta una lista ordenada de datos de la entrada estándar o de un argumento que sea un nombre de archivo (mira la man page de `uniq` para saber más detalles) y, por defecto, elimina los duplicados de la lista. Así, que para estar seguro de que nuestra lista no tiene duplicados (ya sabes, algunos programas con el mismo nombre pueden aparecer tanto en el directorio `/bin` como en `/usr/bin`) añadiremos `uniq` a nuestro pipeline:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

En este ejemplo, usamos `uniq` para eliminar duplicados de la salida del comando `sort`. Si, en lugar de eso, queremos ver la lista de duplicados, añadiremos la opción "-d" a `uniq` así:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc – Muestra el número de líneas, palabras y bytes

El comando `wc` (word count – contador de palabras) se usa para mostrar el número de líneas, palabras y bytes contenidos en un archivo. Por ejemplo:

```
[me@linuxbox ~]$ wc ls-output.txt
7902 64566 503634 ls-output.txt
```

En este caso muestra tres números: líneas, palabras y bytes contenidos en `ls-output.txt`. Como nuestros anteriores comandos, si lo ejecutamos sin argumentos, `wc` acepta la entrada estándar. La opción "-l" limita su salida para mostrar sólo el número de líneas. Añadirlo a un pipeline es una forma útil de contar cosas. Para ver el número de elementos que tenemos en nuestra lista ordenada, podemos hacer esto:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728
```

grep – Imprime líneas que coinciden con un patrón

`grep` es un programa poderoso utilizado para encontrar patrones de texto en los archivos. Se usa así:

```
grep pattern [file...]
```

Cuando `grep` encuentra un "patrón" en el archivo, muestra las líneas que lo contienen. El patrón que `grep` puede encontrar puede ser muy complejo, pero por ahora nos concentraremos en coincidencias simples de texto. Trataremos patrones avanzados, llamados *expresiones regulares* en un capítulo posterior.

Digamos que queremos encontrar todos los archivos en nuestra lista de programas que tengan la palabra "zip" incluida en el nombre. Una búsqueda así debería darnos una idea que algunos de los programas en nuestro sistema que tienen algo que ver con la compresión de archivos. Haríamos ésto:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Hay un par de opciones útiles para `grep`: "-i" que hace que grep ignore las mayúsculas cuando haga la búsqueda (normalmente las búsquedas con sensibles a las mayúsculas) y "-v" que le dice a `grep` que sólo muestre las lineas que no coincidan con el patrón.

head / tail – Muestra la primera/última parte de los archivos

Algunas veces no quieres toda la salida de un comando. Podrías querer sólo las primeras o las últimas líneas. El comando `head` muestra las primeras diez líneas de un archivo y el comando `tail` muestras las diez últimas. Por defecto, ambos comandos muestran diez líneas de texto, pero ésto puede ajustarse con la opción “-n”:

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rwxr-xr-x 1 root root 31316 2007-12-05 08:58 [
-rwxr-xr-x 1 root root 8240 2007-12-09 13:39 411toppm
-rwxr-xr-x 1 root root 111276 2007-11-26 14:27 a2p
-rwxr-xr-x 1 root root 25368 2006-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 root root 5234 2007-06-27 10:56 znew
-rwxr-xr-x 1 root root 691 2005-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root 930 2007-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root 930 2007-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root 6 2008-01-31 05:22 zsoelim -> soelim
```

Puede ser usado en pipelines también:

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

`tail` tiene una opción que nos permite ver los archivos en tiempo real. Ésto es útil para ver el progreso de los archivos de logs tal como se van escribiendo. En el siguiente ejemplo, veremos el archivo messages en `/var/log` (o el archivo `/var/log/syslog` si `messages` no existe). Se requieren privilegios de superusuario para hacerlo en algunas distribuciones Linux, ya que el archivo `/var/log/messages` podría contener información de seguridad:

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb 8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb 8 13:40:05 twin4 dhclient: bound to 192.168.1.4 --
renewal in 1652 seconds.
Feb 8 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox
exported to both 192.168.1.0/24 and twin7.localdomain in
192.168.1.0/24,twin7.localdomain
Feb 8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to
192.168.1.1 port 67
Feb 8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb 8 14:07:37 twin4 dhclient: bound to 192.168.1.4 --
renewal in 1771 seconds.
Feb 8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART
Prefailure Attribute: 8 Seek_Time_Performance changed from
237 to 236
Feb 8 14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox
exported to both 192.168.1.0/24 and twin7.localdomain in
192.168.1.0/24,twin7.localdomain
Feb 8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened
for user me by (uid=0)
```

```
Feb 8 14:25:36 twin4 su(pam_unix)[29279]: session opened
for user root by me(uid=500)
```

Usando la opción “-f”, `tail` continua monitorizando el archivo y cuando se le añaden nuevas líneas, inmediatamente aparecen en la pantalla. Esto continua hasta que pulses `Ctrl-c`.

Tee – Lee de stdin y lo pasa a stdout y a archivos

Siguiendo con nuestra metáfora de fontanería, Linux proporciona un comando llamado `tee` que crea un “soporte” agarrado a nuestra tubería. El programa `tee` lee la entrada estándar y la copia a la salida estándar (permitiendo que los datos continúen bajando por la tubería) y a uno o más archivos. Ésto es útil para capturar el contenido de un pipeline en una fase intermedia del procesamiento. Repetiremos uno de nuestros anteriores ejemplos, esta vez incluyendo `tee` para capturar el listado completo del directorio al archivo `ls.txt` antes de que `grep` filtre el contenido del pipeline:

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Resumiendo

Como siempre, revisa la documentación de cada comando que hemos tratado en este capítulo. Sólo hemos visto su uso más básico. Todos tienen numerosas opciones interesantes. Según vayamos ganando experiencia con Linux, veremos que la función de redirección de la línea de comandos es extremadamente útil para solucionar problemas especializados. Hay muchos comandos que hacen uso de la entrada y salida estándar, y casi todos los programas de la línea de comandos usan el error estándar para mostrar sus mensajes informativos.

Linux tiene que ver con la imaginación

Cuando me piden que explique la diferencia entre Windows y Linux, a menudo uso una analogía con un juguete.

Windows es como una Game Boy. Vas a la tienda y compras una toda brillante y nueva en su caja. La llevas a casa, la enciendes y juegas con ella. Bonitos gráficos, sonidos chulos. Cuando pasa un rato, te cansas del juego que viene con ella y vuelves a la tienda a comprar otro. Este ciclo se repite una y otra vez. Finalmente, vuelves a la tienda y le dices a la persona que está tras el mostrador, “¡Quiero un juego que haga ésto!” sólo para que te digan que ese tipo de juego no existe porque no hay “demanda” en el mercado. Entonces dices, “¡Pero sólo necesito cambiar una cosa! La persona tras el mostrador te dice que no puedes cambiarlo. Los juegos se venden en sus cartuchos. Descubres que tu juguete está limitado a los juegos que otros han decidido que necesitas y ya está.

Linux, al contrario, es como el mecano más grande del mundo. Lo abres y sólo es una gran colección de partes. Un montón de puntillas, tornillos, tuercas, engranajes, poleas, motores todos de acero y algunas sugerencias sobre qué puedes construir. Así que empiezas a jugar con él. Construyes una de las sugerencias y luego otra. Después de un rato descubres que tienes tus propias ideas de qué construir. Nunca más tienes que volver a la tienda, ya que tienes todo lo que necesitas. El mecano se adapta a tu imaginación. Hace lo que tú quieres.

Tu elección de juguetes es, por supuesto, algo personal, así que ¿qué juguete encontrarías más satisfactorio?

Viendo el mundo como lo ve el shell

En este capítulo vamos a ver algo de la "magia" que ocurre en la línea de comandos cuando presionas la tecla enter. Mientras, examinaremos varias características interesantes y complejas del shell, lo haremos con un único comando nuevo:

- `echo` – Muestra una línea de texto

Expansión

Cada vez que escribes un comando y presionas la tecla enter, `bash` realiza varios procesos sobre el texto antes de llevar a cabo el comando. Hemos visto un par de casos de cómo una simple secuencia de caracteres, por ejemplo "*", puede tener mucho significado para el shell. El proceso que hace que ésto ocurra se llama *expansión*. Con la expansión, introduces algo y se expande en otra cosa antes de que el shell actúe sobre ello. Para demostrar qué queremos decir con ésto, echemos un vistazo al comando `echo`. `echo` es una función del shell que realiza una tarea muy simple. Muestra sus argumentos de texto en la salida estándar:

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

Es muy sencillo. Cualquier argumento que pasemos a `echo` se muestra. Probemos otro ejemplo:

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public
Templates
Videos
```

Pero, ¿qué ha pasado? ¿por qué `echo` no ha escrito "*"? Como recordarás de nuestro trabajo con comodines, el carácter "*" significa coincidencia de caracteres en el nombre de archivo, pero lo que no hemos visto en nuestra conversación original es como hace eso el shell. La respuesta sencilla es que el shell expande el "*" en algo más (en este ejemplo, los nombres de los archivos que se encuentran en el directorio de trabajo actual) antes de que el comando echo se ejecute. Cuando presionamos la tecla enter, el shell automáticamente expande todos los caracteres en la linea de comandos antes de que el comando sea ejecutado, por lo que el comando `echo` no ve el "*", sólo su resultado expandido. Sabiendo ésto, podemos ver que `echo` se ha comportado como se esperaba.

Expansión de nombres de archivo

El mecanismo según el cual trabajan los comodines se llama *expansión de nombres de archivo*. Si probamos algunas de las técnicas que hemos empleado en nuestros capítulos anteriores, veremos que son realmente expansiones. Tomemos un directorio home que aparezca de la siguiente forma:

```
[me@linuxbox ~]$ ls
Desktop ls-output.txt Pictures Templates
Documents Music Public Videos
```

podríamos llevar a cabo las siguientes expansiones:

```
[me@linuxbox ~]$ echo D*
Desktop Documents
```

y:

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

o también:

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

y mirando más allá de nuestro directorio home:

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

Expansión de nombres de archivos ocultos

Como sabemos, los nombres de archivo que empiezan por un punto están ocultos. La expansión de nombres de archivo también respeta este comportamiento. Una expansión como:

```
echo *
```

no revela los archivos ocultos.

Podría parecer a primera vista que podríamos incluir archivos ocultos en una expansión comenzando el patrón con un punto, así:

```
echo .*
```

Casi funciona. De todas formas, si examinamos los resultados atentamente, veremos que los nombres "." y ".." también aparecen en los resultados. Como estos nombres se refieren al directorio actual y su directorio padre, usar este patrón probablemente producirá un resultado incorrecto. Podemos verlo si probamos el comando:

```
ls -d .* | less
```

Para ejecutar mejor una expansión de nombres de archivo en esta situación, tenemos que emplear un patrón más específico:

```
echo .[!.]*
```

Este patrón se expande en todos los nombres de archivo que empiecen con un punto, no incluye un segundo punto, seguido de cualquier otro carácter. Ésto funcionará correctamente con la mayoría de archivos ocultos (piensa que todavía no incluirá los nombres de archivo con múltiples puntos al principio). El comando `ls` con la opción -A ("almost all" o "casi todo") proporcionará un listado correcto de los archivos ocultos:

```
ls -A
```

Expansión de la tilde de la ñ

Como recordarás de nuestra introducción al comando `cd`, el carácter virgulilla, o tilde de la ñ ("~" tiene un significado especial. Cuando se usa al principio de una palabra, se expande en el nombre del directorio home del usuario nombrado, o si no se nombra a ningún usuario, en el directorio home del usuario actual:

```
[me@linuxbox ~]$ echo ~
/home/me
```

Si el usuario "foo" tiene una cuenta, entonces:

```
[me@linuxbox ~]$ echo ~foo
/home/foo
```

Expansión aritmética

El shell permite realizar aritmética mediante la expansión. Ésto nos permite usar el prompt del shell como una calculadora:

```
[me@linuxbox ~]$ echo $((2 + 2))
4
```

La expansión aritmética usa la forma:

```
$((expresión))
```

donde expresión es una expresión aritmética consistente en valores y operadores aritméticos.

La expansión aritmética sólo soporta enteros (números enteros sin decimales), pero puede realizar un buen número de operaciones diferentes. Aquí hay unos pocos de los operadores soportados:

Tabla 7-1: Operadores aritméticos

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División (pero recuerda, como la expansión sólo soporta enteros, los resultados serán enteros.)
%	Módulo, que simplemente significa, "resto"
**	Potencia

Los espacios no son significativos en las expresiones aritméticas y las expresiones puede ser anidadas. Por ejemplo, para multiplicar 5 al cuadrado por 3:

```
[me@linuxbox ~]$ echo $(($((5**2)) * 3))75
```

Los paréntesis sencillos pueden ser usados para agrupar subexpresiones. Con esta técnica, podemos reescribir el ejemplo anterior y obtener el mismo resultado usando una única expansión en lugar de dos:

```
[me@linuxbox ~]$ echo $(((5**2) * 3))
75
```

Aquí tenemos un ejemplo usando los operadores división y resto. Fíjate el efecto de la división con enteros:

```
[me@linuxbox ~]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
[me@linuxbox ~]$ echo with $((5%2)) left over.
with 1 left over.
```

La expansión aritmética será tratada con más detalles en el Capítulo 34.

Expansión con llaves

Quizás la expansión más extraña es la llamada *expansión con llaves*. Con ella, puedes crear múltiples cadenas de texto desde un patrón que contenga llaves. Aquí tienes un ejemplo:

```
[me@linuxbox ~]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

Los patrones a expandir con llaves deben contener un prefijo llamado *preámbulo* y un sufijo llamado *postcript*. La expresión entre llaves debe contener una lista de cadenas separadas por comas o un rango de números enteros o caracteres individuales. El patrón no debe contener espacios en blanco. Aquí hay un ejemplo usando un rango de números enteros:

```
[me@linuxbox ~]$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

Los números enteros también pueden tener ceros a la izquierda así:

```
[me@linuxbox ~]$ echo {01..15}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
[me@linuxbox ~]$ echo {001..15}
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015
```

Un rango de letras en orden inverso:

```
[me@linuxbox ~]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Las expansiones con llaves puede ser anidadas:

```
[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

¿Y para qué sirve ésto? La aplicación más común es hacer listas de archivos o directorios a crear. Por ejemplo, si fuésemos fotógrafos con una gran colección de imágenes que queremos organizar en años y meses, la primera cosa que deberíamos hacer es crear una serie de directorios nombrados en formato numérico “Año-Mes”. De esta forma, los directorios se ordenarán cronológicamente. Podríamos escribir la lista completa de directorios, pero sería un montón de trabajo y sería muy fácil equivocarnos. En lugar de eso, podríamos hacer esto:

```
[me@linuxbox ~]$ mkdir Photos
[me@linuxbox ~]$ cd Photos
[me@linuxbox Photos]$ mkdir {2007..2009}-{01..12}
[me@linuxbox Photos]$ ls
2007-01 2007-07 2008-01 2008-07 2009-01 2009-07
2007-02 2007-08 2008-02 2008-08 2009-02 2009-08
```

```
2007-03 2007-09 2008-03 2008-09 2009-03 2009-09
2007-04 2007-10 2008-04 2008-10 2009-04 2009-10
2007-05 2007-11 2008-05 2008-11 2009-05 2009-11
2007-06 2007-12 2008-06 2008-12 2009-06 2009-12
```

¡Muy astuto!

Expansión con parámetros

Sólo vamos a tratar brevemente la expansión con parámetros en este capítulo, pero lo trataremos más extensamente más tarde. Es una función que es más útil en scripts de shell que directamente en la línea de comandos. Muchas de sus capacidades tienen que ver con la capacidad del sistema de almacenar pequeños trozos de datos y dar a cada trozo un nombre. Muchos de esos trozos, mejor llamados variables, están disponibles para que los examines. Por ejemplo, la variable llamada "USER" contiene tu nombre de usuario. Para invocar la expansión con parámetros y revelar el contenido de `USER` deberías hacer ésto:

```
[me@linuxbox ~]$ echo $USERme
```

Para ver una lista de las variables disponibles, prueba esto:

```
[me@linuxbox ~]$ printenv | less
```

Habrás notado que con otros tipos de expansión, si escribes mal un patrón, la expansión no se lleva a cabo y el comando `echo` simplemente mostrará el patrón que has escrito mal. Con la expansión con parámetros, si escribes mal el nombre de la variable, la expansión se realizará, pero dando como resultado una cadena vacía:

```
[me@linuxbox ~]$ echo $SUER
[me@linuxbox ~]$
```

Sustitución de comandos

La sustitución de comandos nos permite usar la salida de un comando como una expansión:

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public
Templates
Videos
```

Una de mis favoritas hace algo como ésto:

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Aquí hemos pasado el resultado de `which cp` como un argumento para el comando `ls`, de esta forma tenemos el listado del programa `cp` sin tener que saber su ruta completa. No estamos limitados a comandos simples sólo. Pipelines completas pueden ser usadas (sólo se muestra una salida parcial):

```
[me@linuxbox ~]$ file $(ls -d /usr/bin/* | grep zip)
/usr/bin/bunzip2: symbolic link to `bzip2'
/usr/bin/bzip2: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs),
for
```

```
GNU/Linux 2.6.9, stripped
/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel
80386,
version 1 (SYSV), dynamically linked (uses shared libs),
for
GNU/Linux 2.6.9, stripped
/usr/bin/funzip: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs),
for
GNU/Linux 2.6.9, stripped
/usr/bin/gpg-zip: Bourne shell script text executable
/usr/bin/gunzip: symbolic link to `../../bin/gunzip'
/usr/bin/gzip: symbolic link to `../../bin/gzip'
/usr/bin/mzip: symbolic link to `mtools'
```

En éste ejemplo, el resultado del pipeline se convierte en la lista de argumentos del comando `file`.

Hay una sintaxis alternativa para la sustitución de comandos en programas de shell antiguos que también es soportada por `bash`. Utiliza *tildes invertidas* en lugar del signo del dólar y los paréntesis:

```
[me@linuxbox ~]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Entrecomillado

Ahora que hemos visto de cuantas formas el shell puede realizar expansiones, es hora de aprender cómo podemos controlarlas. Tomemos un ejemplo:

```
[me@linuxbox ~]$ echo this is a      test
this is a test
```

o:

```
[me@linuxbox ~]$ echo The total is $100.00
The total is 00.00
```

En el primer ejemplo, la *división por palabras* del shell ha eliminado el espacio en blanco de la lista de argumentos del comando `echo`. En el segundo ejemplo, la expansión con parámetros sustituyó una cadena vacía con el valor de “$1” porque era una variable indefinida. El shell proporciona un mecanismo llamado *quoting* (entrecomillado) para suprimir selectivamente expansiones no deseadas.

Comillas dobles

El primer tipo de citas que vamos a ver son las *comillas dobles*. Si colocas un texto dentro de comillas dobles, todos los caracteres especiales utilizados por el shell perderán su significado especial y serán tratados como caracteres ordinarios. Las excepciones son “$”, “\” (barra invertida), y “`” (tilde invertida). Esto significa que la división por palabras, expansión de nombres de archivo, expansión de la tilde de la ñ y la expansión con llaves están suprimidas, pero la expansión con parámetros, la expansión aritmética y la sustitución de comandos sí que funcionarán. Usando comillas dobles, podemos manejar nombres de archivo que contengan espacios en blanco. Digamos que somos la desafortunada víctima de un archivo llamado `two words.txt`. Si tratáramos de usarlo en la línea de comandos, la separación de palabras haría que fuera tratado como dos

argumentos separados en lugar del único argumento que queremos:

```
[me@linuxbox ~]$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

Usando comillas dobles, paramos la separación por palabras y obtenemos el resultado deseado; más aún, incluso podemos reparar el daño causado:

```
[me@linuxbox ~]$ ls -l "two words.txt"
-rw-rw-r-- 1 me me 18 2008-02-20 13:03 two words.txt
[me@linuxbox ~]$ mv "two words.txt" two_words.txt
```

¡Ahí lo tienes! Ahora no tenemos que seguir escribiendo esas malditas comillas dobles.
Recuerda, la expansión con parámetros, la expansión aritmética y la sustitución de comandos siguen funcionando dentro de las comillas dobles:

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"
me 4 February 2008
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

Deberíamos tomarnos un momento para mirar el efecto de las comillas dobles en la sustitución de comandos. Primero miremos un poco más atentamente a cómo funciona la sustitución de palabras. En nuestro ejemplo anterior, vimos como la sustitución de palabras parece eliminar los espacios sobrantes en nuestro texto:

```
[me@linuxbox ~]$ echo this is a      test
this is a test
```

Por defecto, la sustitución de palabras busca la presencia de espacios, tabulaciones y lineas nuevas (caracteres de inicio de línea) y las trata como *delimitadores* entre palabras. Ésto significa que los espacios sin comillas, tabuladores y nuevas líneas no se consideran parte del texto. Sólo sirven como separadores. Como separan las palabras en diferentes argumentos, nuestra línea de comandos de ejemplo contiene un comando seguido de diferentes argumentos. Si añadimos comillas dobles:

```
[me@linuxbox ~]$ echo "this is a       test"
this is a      test
```

la separación de palabras se suprime y los espacios en blanco no se tratan como separadores, en lugar de eso pasan a ser parte del argumento. Una vez que añadimos las comillas dobles, nuestra línea de comandos contiene un comando seguido de un único argumento.

El hecho de que las líneas nuevas sean consideradas como separadores por el mecanismo de separación de palabras provoca un interesante, aunque sutil, efecto en la sustitución de comandos. Considera lo siguiente:

```
[me@linuxbox ~]$ echo $(cal)
February 2008 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox ~]$ echo "$(cal)"
February 2008
```

```
Su Mo Tu We Th Fr Sa
                1  2
       3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

En el primer caso, la sustitución de comandos sin comillas resulta en una línea de comandos que contiene 38 argumentos. En el segundo, una linea de comandos con un argumento que incluye los espacios en blanco y las nuevas líneas.

Comillas simples

Si necesitamos suprimir *todas* las expansiones, usamos *comillas simples*. A continuación vemos una comparación entre sin comillas, comillas dobles y comillas simples:

```
[me@linuxbox ~]$ echo text ~/*.txt {a,b} $(echo foo) $
((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox ~]$ echo "text ~/*.txt {a,b} $(echo foo) $
((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox ~]$ echo 'text ~/*.txt {a,b} $(echo foo) $
((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

Como podemos ver, con cada nivel sucesivo de entrecomillado, se van suprimiendo más expansiones.

Caracteres de escape

Algunas veces sólo queremos entrecomillar un único carácter. Para hacerlo, podemos preceder un carácter con una barra invertida, que en este contexto se llama *carácter de escape*. A menudo se hace dentro de las comillas dobles para prevenir selectivamente una expansión:

```
[me@linuxbox ~]$ echo "The balance for user $USER is: \
$5.00"
The balance for user me is: $5.00
```

También es común usar caracteres de escape para eliminar el significado especial de un carácter en un nombre de archivo. Por ejemplo, es posible usar caracteres en nombres de archivo que normalmente tienen un significado especial para el shell. Ésto incluye “$”, “!”, “&”, “ “, y otros. Para incluir un carácter especial en un nombre de archivo puedes hacer ésto:

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

Para permitir que la barra invertida aparezca, la “escapamos” escribiendo “\\”. Fíjate que dentro de las comillas simples, la barra invertida pierde su significado especial y se trata como un carácter ordinario.

Secuencias de escape con la barra invertida

Además de su rol como carácter de escape, la barra invertida también se usa como parte de una notación para representar ciertos caracteres especiales llamados *códigos de control*. Los primeros 32

caracteres en el esquema de código ASCII se usan para transmitir comandos a dispositivos de la familia de los teletipos. Algunos de estos códigos son familiares (tabulador, salto de línea y salto de párrafo), mientras que otros no lo son (nulo, fin de la transmisión, y entendido).

Secuencia de escape	Significado
`\a`	Tono (“Alerta” - hace que el ordenador pite)
`\b`	Retroceder un espacio
`\n`	Nueva línea. En sistemas como-Unix, produce un salto de línea.
`\r`	Retorno de carro.
`\t`	Tabulación

La tabla anterior lista algunos de las secuencias de caracteres de escape más comunes. La idea detrás de esta representación usando la barra invertida se originó en la programación en lenguaje C y ha sido adoptada por otros muchos, incluido el shell.

Añadiendo la opción “-e” a `echo` activaremos la interpretación de las secuencias de escape. También puedes colocarlos dentro de `$' '`. Aquí, usando el comando `sleep`, un programa simple que sólo espera a que le digamos un número específico de segundos y luego se cierra, podemos crear un primitivo cronómetro de cuenta atrás:

sleep 10; echo -e "Time's up\a"

También podríamos hacer ésto:

sleep 10; echo "Time's up" $'\a'

Resumiendo

A medida que avancemos en el uso del shell, encontraremos que las expansiones y entrecomillados se usarán con mayor frecuencia, así que cobra sentido tener un buen entendimiento de la forma en que funcionan. De hecho, se podría decir que son el aspecto más importante a aprender sobre el shell. Sin un entendimiento adecuado de la expansión, el shell siempre será una fuente de misterio y confusión, y perderemos mucho de su potencial.

Para saber más

La man page de `bash` tiene secciones principales tanto de expansión como de entrecomillado que cubren estos asuntos de una manera más formal.

El *Manual de Referencia de Bash* también contiene capítulos de expansión y entrecomillado: http://www.gnu.org/software/bash/manual/bashref.html

Trucos avanzados del teclado

A menudo bromeo y describo Unix como “el sistema operativo para la gente que le gusta escribir”. De acuerdo, el hecho es que incluso tenga una línea de comandos es una prueba de ello. Pero a los usuarios de la línea de comandos no les gusta tanto escribir. ¿Por qué entonces muchos comandos tiene nombres cortos como `cp`, `ls`, `mv` y `rm`? De hecho, uno de los objetivos más preciados de la línea de comandos es la pereza; hacer la mayor cantidad de trabajo con el menor número de pulsaciones. Otro objetivo es no tener que despegar nunca los dedos del teclado, no tener que coger nunca el ratón. En éste capítulo, veremos las funciones de `bash` que hacen que usemos el teclado

más rápido y más eficientemente.
Los siguientes comandos harán su aparición:

- `clear` – Limpia la pantalla
- `history` – Muestra los contenidos de la lista de historia

Edición de la línea de comandos

`bash` usa una librería (una colección compartida de rutinas que pueden utilizar diferentes programas) llamada *Readline* para implementar la edición de la línea de comandos. Ya hemos visto algo de ésto. Sabemos, por ejemplo, que las teclas de las flechas mueven el cursor, pero hay muchas más características. Piensa en ello como herramientas adicionales que podemos emplear en nuestro trabajo. No es importante aprenderlo todo sobre ellas, pero muchas son muy útiles. Cójelas y elígelas como quieras.

Nota: Muchas de las siguientes secuencias de teclas (particularmente aquellas que usan la tecla `Alt`) podrían ser interceptadas por el GUI para otras funciones. Todas las secuencias de teclas deberían funcionar adecuadamente cuando usamos una consola virtual.

Movimiento del cursor

La siguiente tabla lista las teclas usadas para mover el cursor:

Tabla 8-1: Comandos de movimiento del cursor

Key	**Acción**
`Ctrl -a`	Mueve el cursor al principio de la línea
`Ctrl -e`	Mueve el cursor al final de la línea
`Ctrl -f`	Mueve el cursor un carácter hacia delante; igual que la flecha derecha
`Ctrl -b`	Mueve el cursor un carácter hacia atrás; igual que la flecha izquierda
`Alt -f`	Mueve el cursor hacia delante una palabra.
`Alt -b`	Mueve el cursor hacia detrás una palabra.
`Ctrl -l`	Limpia la pantalla y mueve el cursor a la esquina superior izquierda. El comando `clear` hace lo mismo.

Modificando el texto

La tabla 8 – 2 lista los comandos de teclado que se usan para editar caracteres en la línea de comandos.

Tabla 8 -2: Comandos de edición de texto

Comando	Acción
`Ctrl -d`	Borra un carácter en la localización del cursor
`Ctrl -t`	Traspasa (intercambia) el carácter en la localización del cursor con el que le precede
`Alt -t`	Traspasa la palabra en la localización del cursor con la que le precede
`Alt -l`	Convierte los caracteres desde la localización del cursor hasta el final de la palabra a minúsculas
`Alt -u`	Convierte los caracteres desde la localización del cursor hasta el final de la palabra a mayúsculas

Cortar y pegar (Killing and Yanking) texto

La documentación Readline utiliza los términos *killing* (matar) y *yanking* (tirar) para referirse a lo que comúnmente denominamos cortar y pegar. Los elementos que se cortan son almacenados en un buffer llamado *kill-ring*.

Tabla 8-3: Comando para cortar y pegar

Comando	Acción
`Ctrl -k`	Kill (corta) el texto desde la localización del cursor al final de la línea.
`Ctrl -u`	Kill (corta) el texto desde la localización del cursor hasta el principio de la línea.
`Alt -d`	Kill (corta) el texto desde la localización del curso hasta el final de la palabra actual.
`Alt -Retroceso`	Kill (corta) el texto desde la localización del cursor hasta el principio de la palabra actual. Si el cursor está al principio de una palabra, corta la palabra anterior.
`Ctrl -y`	Yank (pega) texto del kill-ring y lo coloca en la localización del cursor

La Meta tecla

Si te aventuras en la documentación Readline, que puedes encontrar en la sección READLINE de la man page de `bash`, encontrarás el término "meta key". En los teclados modernos te dirige a la tecla `Alt` pero no siempre es así.

Volviendo a los tiempos oscuros (antes de los Pcs pero después de Unix) no todo el mundo tenía su propio ordenador. Lo que podían tener era un dispositivo llamado terminal. Un *terminal* era un dispositivo de comunicación que con contaba con una pantalla para mostrar texto y únicamente la electrónica suficiente dentro para mostrar los caracteres y mover el cursor. Estaba conectado (normalmente con un cable serie) a una computadora mayor o a la red de comunicación de una computadora mayor. Había muchas marcas diferentes de terminales y todas tenían configuraciones de pantalla y teclado diferentes. Como todas tendían al menos a entender ASCII, los desarrolladores de software querían escribir aplicaciones portables reducidas al mínimo común denominador. Los sistemas Unix tienen una forma muy elaborada de tratar con los terminales y sus pantallas. Como los desarrolladores de Readline no podían estar seguros de la presencia de una tecla de control extra, se inventaron una y la llamaron "meta". Igual que la tecla Alt sirve como tecla meta en los teclados modernos, puedes también pulsar y soltar la tecla Esc para tener el mismo efecto que pulsando la tecla `Alt` si todavía estás usando un terminal (¡Lo que aún puedes hacer en Linux!).

Completado

Otra forma en que el shell puede ayudarte es mediante un mecanismo llamado *completado*. El completado ocurre cuando presionas la tecla tabulador mientras escribes un comando. Veamos como funciona. Dado un directorio home que aparece así:

```
[me@linuxbox ~]$ ls
Desktop ls-output.txt Pictures Templates Videos
Documents Music Public
```

Prueba escribiendo lo siguiente pero **no pulses la tecla `Enter`:**

```
[me@linuxbox ~]$ ls l
```

Ahora pulsa la tecla tabulador:

```
[me@linuxbox ~]$ ls ls-output.txt
```

¿Ves como el shell completa la línea por tí? Probemos otra vez. De nuevo, no pulses `Enter`:

```
[me@linuxbox ~]$ ls D
```

Pulsa tab:

```
[me@linuxbox ~]$ ls D
```

No completa, sólo da un pitido. Esto ocurre porque "D" coincide con más de una entrada en el directorio. Para que el completado funcione, el "indicio" que le des no debe ser ambiguo. Si seguimos:

```
[me@linuxbox ~]$ ls Do
```

Y pulsamos tab:

```
[me@linuxbox ~]$ ls Documents
```

El completado funciona.

Aunque este ejemplo muestra el completado de nombres de archivo, que es su uso más común, el completado también funciona con variables (si el principio de la palabra es un "$"), nombres de usuario (si la palabra empieza con "~"), comandos (si la palabra es la primera en la línea.) y nombres de equipo (si la palabra empieza con "@"). El completado de nombres de equipo sólo funciona para los nombres de equipo listados en `/etc/hosts`.

Hay un número de secuencias de meta teclas y teclas de control que están asociadas al completado:

Tabla 8-4: Comandos de completado

Tecla	Acción
`Alt-?`	Muestra una lista de posibles completados. En la mayoría de los sistemas también puedes hacerlo pulsando la tecla tabulador una segunda vez, que es mucho más fácil.
`Alt-*`	Introduce todos los completados posibles. Ésto es útil cuando quieres usar más de un resultado posible.

Hay algunos más que me parecen bastante extraños. Puedes ver una lista en la man page de `bash` bajo "READLINE".

Completado programable

Versiones recientes de `bash` tienen una característica llamada *completado programable*. El completado programable te permite (o mejor dicho, el proveedor de tu distribución te permite) añadir reglas de completado adicionales. Normalmente ésto se hace para añadir soporte para aplicaciones específicas. Por ejemplo es posible añadir completados para la lista de opciones de un comando o para buscar tipos de archivo en concreto que soporte una aplicación. Ubuntu tiene una gran colección definida por defecto. El completado programable se implementa mediante funciones de shell, un tipo de mini scripts de shell que veremos en próximos capítulos. Si eres curioso, prueba:

set | less

y mira si puedes encontrarlos. No todas las distribuciones las incluyen por defecto.

Usando el historial

Como descubrimos en el Capítulo 1, `bash` mantiene un historial de los comandos que hemos introducido. Esta lista de comandos se guarda en nuestro directorio home en un archivo llamado `.bash_history`. El historial es un recurso útil para reducir la cantidad de pulsaciones que tenemos que hacer, especialmente cuando lo combinamos con la edición en la línea de comandos.

Buscando en el Historial

En cualquier momento, podemos ver el contenido del historial así:

```
[me@linuxbox ~]$ history | less
```

Por defecto, bash almacena los últimos 500 comandos que has utilizado. Veremos como ajustar este valor en un capítulo posterior. Digamos que queremos encontrar los comandos que hemos usado para listar `/usr/bin`. Una de las formas en que podríamos hacerlo sería así:

```
[me@linuxbox ~]$ history | grep /usr/bin
```

Y digamos que entre los resultados tenemos una línea que contiene un comando interesante como éste:

```
88 ls -l /usr/bin > ls-output.txt
```

El número "88" es el número de línea del comando en el historial. Podríamos usarlo inmediatamente mediante otro tipo de expansión llamada *expansión del historial*. Para utilizar la línea que hemos descubierto podríamos hacer ésto:

```
[me@linuxbox ~]$ !88
```

`bash` expandirá "!88!" en el contenido de la octogésimo octava línea del historial. Hay otras formas de expandir el historial que veremos un poco más tarde.

`bash` cuenta también con la capacidad de buscar en el historial incrementalmente. Ésto significa que podemos decirle a `bash` que busque en el historial mientras vamos introduciendo caracteres, con cada carácter adicional más refinada será nuestra búsqueda. Para empezar una búsqueda incremental pulsamos `Ctrl-r` seguido del texto que estamos buscando. Cuando lo encuentras, puedes pulsar `Enter` para ejecutar el comando o pulsar `Ctrl-j` para copiar la línea del historial a la actual linea de la consola. Para buscar la siguiente coincidencia del texto (en el sentido "hacia arriba" en el historial), pulsamos `Ctrl-r` de nuevo. Para dejar de buscar, pulsamos `Ctrl-g` o `Ctrl-c`. Aquí lo vemos en acción:

```
[me@linuxbox ~]$
```

Primero pulsamos `Ctrl-r`:

```
(reverse-i-search)`':
```

El prompt cambia para indicar que estamos realizando un búsqueda incremental inversa. Es "inversa" porque estamos buscando desde "ahora" a algún momento en el pasado. Ahora, empezamos a escribir nuestro texto de búsqueda. En este ejemplo "/usr/bin":

```
(reverse-i-search)`/usr/bin': ls -l /usr/bin > ls-
```

```
output.txt
```

Inmediatamente, la búsqueda nos devuelve nuestro resultado. Con nuestro resultado podemos ejecutar el comando presionando `Enter`, o copiarlo a nuestra consola de comandos para editarlo después presionando `Ctrl-j`. Vamos a copiarlo. Pulsamos `Ctrl-j`:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Nuestro prompt de shell vuelve y nuestra línea de comandos está cargada y ¡lista para la acción! La siguiente tabla lista alguna de los atajos de teclas usados para manipular el historial:

Tabla 8-5: Comandos del historial

Tecla	Acción
`Ctrl-p`	Se mueve a la anterior entrada del historial. Misma acción que la flecha hacia arriba
`Ctrl-n`	Se mueve al la siguiente entrada del historial. Misma acción que la flecha hacia abajo
`Alt-<`	Se mueve al principio (arriba) del historial.
`Alt->`	Se mueve al final (abajo) del historial, p.ej., la actual línea de comandos.
`Ctrl-r`	Búsqueda incremental inversa. Busca incrementalmente desde el actual comando hacia arriba en el historial.
`Alt-p`	Búsqueda inversa, no incremental. Con esta combinación de teclas, escribes las palabras a buscar y presionas enter antes de que la búsqueda se realice.
`Alt-n`	Búsqueda hacia delante, no incremental.
`Ctrl-o`	Ejecuta el elemento actual en el historial y avanza al siguiente. Ésto es práctico si estás intentando reejecutar una secuencia de comandos del historial.

Expansión del historial

El shell ofrece un tipo especializado de expansión para elementos del historial usando el carácter "!". Ya hemos visto cómo el signo de exclamación puede ir seguido por un número para insertar una entrada del historial. Hay otras aplicaciones de la expansión:

Tabla 8-6: Comandos de la expansión del historial

Secuencia	Acción
`!!`	Repite el último comando. Probablemente es más fácil pulsar la flecha y luego enter.
`!`*`número`*	Repite el elemento del historial número.
`!`*`texto`*	Repite el último elemento del historial que empiece con texto.
`!?texto`	Repite el último elemento del historial que contenga texto.

Quisiera aconsejarte no usar las formas "!texto" y "!?texto!" a no ser que estés completamente seguro del contenido de los elementos del historial.

Hay muchos más elementos disponibles en el mecanismo de expansión del historial, pero es un asunto demasiado arcaico y nuestras cabezas podrían explotar si continuamos. La sección HISTORY EXPANSION de la man page de `bash` ahonda en todos los detalles macabros. ¡Eres libre de explorarlo!

script

Además de la función del historial de comandos en bash, la mayoría de las distribuciones Linux

incluyen un programa llamado `script` que se puede utilizar para grabar una sesión de shell completa y almacenarla en un archivo. La sintaxis básica del comando es:

```
script [archivo]
```

donde archivo es el nombre del archivo utilizado para almacenar la grabación. Si no se especifica ningún archivo, se usa el archivo `typescript`. Mira la man page de `script` para ver una lista completa de las opciones y funciones del programa.

Resumiendo

En este capítulo hemos tratado algunos de los trucos de teclado que el shell ofrece para ayudar a los duros pulsadores de teclas reducir su carga de trabajo. Sospecho que según pase el tiempo y te familiarices más con la línea de comandos, volverás a este capítulo para recoger más de estos trucos. Por ahora, considéralos opcionales y potencialmente útiles.

Para saber más

- Wikipedia tiene un buen artículo sobre los terminales:

  http://en.wikipedia.org/wiki/Computer_terminal

Permisos

Los sistemas operativos tradicionales de Unix difieren de los tradicionales de MS-DOS en que no sólo son sistemas *multitarea*, sino que tembién son sistemas *multiusuario*.

¿Qué significa ésto exactamente? Significa que más de una persona puede estar usando el ordenador al mismo tiempo. Aunque que un ordenador típico tendrá sólo un teclado y un monitor, podrá ser usado por más de un usuario. Por ejemplo, si un ordenador está conectado a una red o a Internet, los usuarios remotos pueden acceder via `ssh` (secure shell) y utilizar el ordenador. De hecho, los usuarios remotos pueden ejecutar aplicaciones gráficas y hacer que la salida gráfica aparezca en una pantalla remota. El Sistema X Windows soporta ésto como una parte de su diseño básico.

La capacidad multiusuario de Linux no es una innovación reciente, sino una característica que está profundamente integrada en el diseño del sistema operativo. Considerando el entorno en el que Unix fue creado, ésto tiene todo el sentido. Hace años, antes de que los ordenadores fueran "personales", eran grandes, caros, y centralizados. Un ordenador típico de una universidad, por ejemplo, consistía en un gran ordenador central situado en un edificio y terminales localizados a lo largo del campus, cada uno conectado al gran ordenador central. El ordenador tenía que soportar muchos usuarios al mismo tiempo.

Para hacer ésto práctico, el sistema tenía que ser ideado para proteger a los usuarios unos de otros. Después de todo, las acciones que de un usuario no podían permitirle estropear el ordenador, ni un usuario podía interferir en los archivos que pertenecía a otro usuario.

En este capítulo vamos a ver esta parte esencial de la seguridad del sistema y presentaremos los siguientes comandos:

- `id` – Muestra la identidad del usuario
- `chmod` – Cambia el modo de un archivo
- `umask` – Establece los permisos por defecto
- `su` – Ejecuta un shell como otro usuario
- `sudo` – Ejecuta un comando como otro usuario
- `chown` – Cambia el propietario de un archivo
- `chgrp` – Cambia la propiedad de grupo de un archivo
- `passwd` – Cambia la contraseña de un usuario

Propietarios, miembros del grupo, y todos los demás

Cuando estabamos explorando en sistema allá por el capítulo 3, pudimos encontrar un problema cuando intentábamos examinar un archivo como `/etc/shadow`:

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow: regular file, no read permission
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow: Permission denied
```

La razón de este mensaje de error es que, como usuarios normales, no tenemos permiso para leer este archivo.

En el modelo de seguridad de Unix, un usuario puede *poseer* archivos y directorios. Cuando un usuario posee un archivo o directorio, el usuario tiene control sobre su acceso. Los usuarios pueden, sucesivamente, pertenecer a un *grupo* consistente en uno o más usuarios a quienes se le ha dado acceso a archivos y directorios por sus propietarios. Además de conceder acceso a un grupo, un propietario puede también conceder algún tipo de derechos de acceso a todo el mundo, a quienes en el lenguaje de Unix nos referimos como el mundo. Para encontrar información sobre tu identidad, usa el comando `id`:

```
[me@linuxbox ~]$ id
uid=500(me) gid=500(me) groups=500(me)
```

Veamos la salida. Cuando se crea una cuenta de usuario, al usuario se le asigna un número llamado ID de usuario o uid que es, por el bien de los humanos, asignado a un nombre de usuario. Al usuario se le asigna un *ID de grupo primario* o *gid* y puede pertenecer a grupos adicionales. El ejemplo anterior es de un sistema Fedora. En otros sistemas, como Ubuntu, la salida puede ser un poco diferente:

```
[me@linuxbox ~]$ id
uid=1000(me) gid=1000(me)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30
(dip),44(v
ideo),46(plugdev),108(lpadmin),114(admin),1000(me)
```

Como podemos ver, los números uid y gid son diferentes. Ésto simplemente es porque Fedora empieza numerando las cuentas de los usuarios normales desde el 500, mientras que Ubuntu empieza en el 1000. También podemos ver que el usuario de Ubuntu pertenece a muchos más grupos. Ésto tiene que ver con la forma en que Ubuntu maneja los privilegios para los dispositivos y servicios del sistema.

Pero ¿de dónde viene esta información? Como tantas otras muchas cosas en Linux, de un par de archivos de texto. Las cuentas de usuario está definidas en el archivo `/etc/passwd` y los grupos

están definidos en el archivo `/etc/group`. Cuando las cuentas y los grupos de usuarios son creados, estos archivos son modificados junto con archivo `/etc/shadow` que guarda la información de las contraseñas de los usuarios. Para cada cuenta de usuario, el archivo `/etc/passwd` define el nombre de usuario (login), uid, gid, el nombre real de la cuenta, el directorio home y el shell de login. Si examinas el contenido de `/etc/passwd` y `/etc/group`, verás que además de las cuentas de los usuarios normales, hay cuentas para el superusuario (uid 0) y varios otros usuarios del sistema.

En el próximo capítulo, veremos procesos, y verás que algunos de esos otros usuarios están, de hecho, muy ocupados.

Mientras que muchos sistemas como-Unix asignan los usuarios normales a un grupo común como “users”, la práctica en los Linux modernos es crear un único grupo, con un único usuario y el mismo nombre que el usuario. Ésto hace más fácil asignar ciertos tipos de permisos.

Leer, escribir y ejecutar

Los derechos de acceso a archivos y directorios se definen en términos de derechos de lectura, escritura y ejecución. Si miramos la salida del comando `ls`, podemos obtener alguna pista de cómo se implementan:

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me me 0 2008-03-06 14:52 foo.txt
```

Los primeros diez caracteres de la lista son los *atributos del archivo*. El primero de éstos caracteres es el *tipo de archivo*. Aquí tenemos los tipos de archivo que se ven más a menudo (hay otros, menos comunes):

Tabla 9-1: Tipos de archivo

Atributo	Tipo de archivo
-	Archivo normal
d	Directorio
l	Enlace simbólico. Fíjate que con los enlaces simbólicos, el resto de los atributos del archivo son siempre “rwxrwxrwx” y que son valores de relleno. Los atributos reales son los del archivo al que el enlace simbólico apunta.
c	*Archivo de carácter especial*. Este tipo de archivo se refiere a un dispositivo que soporta datos como una cadena de bytes, como un terminal o un modem.
b	*Archivo de bloque especial*. Este tipo de archivo se refiere a un dispositivo que soporta datos en bloques, como un disco duro o una unidad de CD.

Los nueve caracteres restantes de los atributos del archivo, se llaman *modo de archivo*, y representan los permisos de lectura, escritura y ejecución del propietario de archivo, del grupo del propietario del archivo y del resto de usuarios:

Propietario	Grupo	Mundo
`rwx`	`rwx`	`rwx`

Cuando establecemos los atributos de modo `r`, `w` y `x` tienen el siguiente efecto sobre los archivos y directorios:

Tabla 9-2: Atributos de permisos

Atributo	Archivos	Directorios
`r`	Permite que un archivo sea abierto y leido.	Permite que el contenido del directorio sea listado si el atributo de ejecución también está configurado.
`w`	Permite que un archivo sea escrito o truncado, sin embargo este atributo no permite renombrar o borrar los archivos. La capacidad de borrar o renombrar archivos viene determinada por los atributos del directorio.	Permite crear, borrar y renombrar archivos dentro de un directorio si el atributo de ejecución también está establecido.
`x`	Permite que un archivo sea tratado como un programa y ejecutarlo. Los archivos de programas escritos en lenguajes de script deben ser también configurados como legibles para ser ejecutados.	Permite entrar en el directorio, p.ej., cd *directorio*

Aquí tenemos algunos ejemplos de configuraciones de atributos de archivo:

Tabla 9-3: Ejemplos de atributos de permisos

Atributos de archivo	Significado
`-rwx------`	Un archivo normal que se puede leer, escribir y ejecutar por el propietario del archivo. Nadie más tiene acceso.
`-rw-------`	Un archivo normal que se puede leer y escribir por el propietario del archivo. Nadie más tiene acceso.
`-rw-r—r--`	Un archivo normal que se puede leer y escribir por el propietario del archivo. Los miembros del grupo del propietario del archivo puede leerlo. El archivo lo puede leer todo el mundo.
`-rwxr-xr-x`	Un archivo normal que lo puede leer, escribir y ejecutar el propietario del archivo. El archivo puede ser leído y ejecutado por todos los demás.
`-rw-rw----`	Un archivo normal que lo puede leer y escribir el propietario del archivo y los miembros de su grupo nada más.
`lrwxrwxrwx`	Un enlace simbólico. Todos los enlaces simbólicos tienen permisos de relleno. Los permisos reales se guardan en el archivo real al que apunta el enlace simbólico.
`drwxrwx---`	Un directorio. El propietario y los miembros de su grupo pueden entrar en el directorio y crear, renombrar y borrar archivos dentro del directorio.
`drwxr-x---`	Un directorio. El propietario puede entrar en el directorio y crear, renombrar y borrar archivos dentro del directorio. Los miembros del grupo del propietario pueden entrar en el directorio pero no pueden crear, borrar o renombrar archivos.

chmod – Cambiando el modo de archivo

Para cambiar el modo (los permisos) de un archivo o un directorio, se utiliza el comando `chmod`. Ten en cuenta que sólo el propietario del archivo o el superusuario pueden cambiar el modo de un archivo o un directorio. `chmod` soporta dos formas distintas de especificar los cambios de modo: representación en números octales o representación simbólica. Veremos la representación en números octales primero.

Con la notación octal usamos números octales para especificar el patrón de los permisos que queremos. Como cada dígito en un número octal representa tres dígitos binarios, esto coincide muy bien con la forma de almacenar el modo de archivo. Esta tabla muestra lo que queremos decir:

Tabla 9-4: Modos de archivo en Binario y Octal

Octal	**Binario**	**Modo de archivo**
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

Usando tres dígitos octales, podemos establecer el modo de archivo para el propietario, el grupo del propietario y el mundo:

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me me 0 2008-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 me me 0 2008-03-06 14:52 foo.txt
```

Pasándole el argumento "600", podemos establecer los permisos del propietario para leer y escribir mientras borramos todos los permisos para el grupo del propietario y el mundo. Aunque recordar la correspondencia entre octal y binario puede parecer un inconveniente, normalmente sólo tendrás que usar algunos muy comunes: `7(rwx)`, `6(rw-)`, `5(r-x)`, `4(r--)`, y `0(---)`.

`chmod` también soporta notación simbólica para especificar los modos de archivo. La notación simbólica se divide en tres partes: a quien afecta el cambio, que operación se realizarán, y que permisos se establecerán. Para especificar quien se ve afectado se usa una combinación de los caracteres "u", "g", "o" y "a" de la siguiente manera:

Tabla 9-5: Notación simbólica de chmod

Símbolo	Significado
u	Abreviatura de "usuario" pero significa el propietario del archivo o el directorio.
g	El grupo del propietario.
o	Abreviatura de "otros", pero se refiere al mundo.
a	Abreviatura de "all" (todos). Es la combinación de "u", "g" y "o"

Si no se especifica ningún carácter, se asume "all". La operación puede ser un "+" indicando que un permiso ha de ser añadido, un "-" indicando que un permiso ha de ser retirado, o un "=" indicando que sólo los permisos especificados deben ser aplicados y todos los demás han de ser eliminados.

Los permisos se especifican con los caracteres "r", "w" y "x". Aquí tenemos algunos ejemplos de la notación simbólica:

Tabla 9-6 Ejemplos de notación simbólica de chmod

Notación	Significado
`u+x`	Añade permisos de ejecución para el propietario
`u-x`	Elimina permisos de ejecución del propietario
`+x`	Añade permisos de ejecución para el propietario, el grupo y el mundo. Equivalente a `a+x`
`o-rw`	Elimina los permisos de lectura y escritura para todos incluyendo el propietario y el grupo del propietario.
`go=rw`	Configura al grupo del propietario y todo el mundo incluido del propietario para que tengan permisos de lectura y escritura. Si el grupo del propietario o el mundo previamente tenían permisos de ejecución, los elimina.
`u+x, go=rx`	Añade permisos de ejecución para el propietario y establece los permisos para el grupo y para otros de lectura y ejecución. Múltiples especificaciones pueden ser separadas por comas.

Algunos prefieren usar la notación octal, otros prefieren la simbólica. La notación simbólica ofrece la ventaja de permitirte establecer un atributo individual sin molestar a los otros.

Échale un vistazo a la man page de chmod para más detalles y una lista de opciones. Un aviso de precaución acerca de la opción “--recursive”: actúa tanto en archivos como en directorios, así que no es tan útil como uno esperaría, raras veces queremos que archivos y directorios tengan los mismos permisos.

¿Qué diablos es un octal?

Octal (base 8), y su primo, *hexadecimal* (base 16) son sistemas numéricos que se utilizan a menudo para expresar números en ordenadores. Nosotros los humanos, debido al hecho de que (al menos la mayoría de nosotros) hemos nacido con diez dedos, contamos usando un sistema numérico en base 10. Los ordenadores, por otra parte, han nacido sólo con un dedo y por tanto todo lo cuentan en sistema binario (base 2). Su sistema numérico sólo tiene dos numerales, 0 y 1. Así que contar en binario tiene esta pinta:
0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011...
En octal, se cuenta con los números de cero al siete, así:
0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21...
Contar en hexadecimal usa los números de cero a nueve más las letras "A" a "F":
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...
Mientras que podemos ver el sentido del binario (ya que los ordenadores sólo tienen un dedo), ¿para que sirven los octales y los hexadecimales? La respuesta tiene que ver con la conveniencia humana. Muchas veces, pequeños trozos de datos se representan en los ordenadores como *patrones de bits*. Tomemos como ejemplo un color RGB. En la mayoría de las pantallas de ordenador, cada pixel se compone de tres componentes de color: ocho bits para el rojo, ocho bits para el verde y ocho bits para el azul. Un bonito azul claro podría ser un número de 24 dígitos:
010000110110111111001101
¿Te gustaría leer y escribir este tipo de números todos los días? Creo que no. Aquí es donde otro sistema numérico nos puede ayudar. Cada dígito en hexadecimal representa cuatro dígitos en binario. En octal, cada dígito representa tres dígitos binarios. Así que nuestro azul claro de 24 dígitos podría reducirse a un número hexadecimal de seis dígitos:
436FCD
Cómo los dígitos en el número hexadecimal "se alinean" con los dígitos del número binario, podemos ver que el componente rojo de nuestro color es 43, el verde 6F, y el azul CD.

Hoy en día, la notación hexadecimal (a menudo llamada "hex") es más común que la octal, pero como veremos pronto, la capacidad de los octales para expresar tres bits de binario será muy útil...

Configurando el modo de un archivo con la GUI

Ahora que hemos visto como se configuran los permisos en archivos y directorios, podemos entender mejor los diálogos de permisos en la GUI. Tanto en Nautilus (GNOME) como en Konqueror (KDE), pulsar con el botón derecho en el icono de un archivo o un directorio nos mostrará un cuadro de diálogo de propiedades. Aquí tenemos un ejemplo de KDE 3,5:

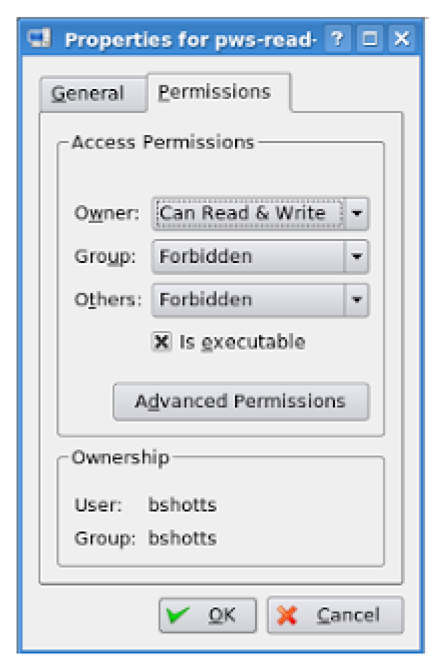


Figura 2: Cuadro de diálogo de propiedades de un archivo en KDE 3.5

Aquí podemos ver la configuración para el propietario, el grupo y el mundo. En KDE, pulsando el botón "Advanced Permissions" (Permisos avanzados), nos mostrará un nuevo diálogo que nos permitirá establecer cada uno de los atributos de modo individualmente. ¡Otra victoria a la comprensión proporcionada por la línea de comandos!

umask – Establecer los permisos por defecto

El comando `umask` controla los permisos por defecto dados a un archivo cuando éste es creado. Usa notación octal para expresar una *máscara* de bits que serán eliminados de los atributos del modo de un archivo. Echemos un vistazo:

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ umask
0002
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me me 0 2008-03-06 14:53 foo.txt
```

Primero hemos eliminado cualquier copia antigua de `foo.txt` para asegurarnos de que empezamos de cero. A continuación, hemos ejecutado el comando `umask` sin argumentos para ver el valor actual. Ha respondido con el valor `0002` (el valor `0022` es otro valor por defecto muy común), que es la representación octal de nuestra máscara. Luego hemos creado una nueva instancia de `foo.txt` y hemos observado sus permisos.

Podemos ver que tanto el propietario como el grupo tienen permisos de lectura y escritura, mientras que los demás sólo tienen permisos de lectura. La razón por la que el mundo no tiene permisos de escritura es por el valor de la máscara. Repitamos nuestro ejemplo, esta vez estableciendo la máscara nosotros mismos:

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-rw- 1 me me 0 2008-03-06 14:58 foo.txt
```

Cuando establecemos la máscara como `0000` (en la práctica la desactivamos), podemos ver que el archivo ahora es editable por el mundo. Para entender como funciona, tenemos que mirar a los números octales de nuevo. Si tomamos la máscara y la expandimos a binario, y luego la comparamos con los atributos, podemos ver lo que sucede:

Modo de archivo original	`--- rw- rw- rw-`
Máscara	`000 000 000 010`
Resultado	`--- rw- rw- r--`

Olvida por el momento los ceros a la izquierda (volveremos a ellos en un minuto) y observa que donde aparece el 1 en nuestra máscara, el atributo se elimina – en este caso, los permisos de escritura del mundo. Ésto es lo que hace la máscara. Donde aparezca un 1 en el valor binario de la máscara, un atributo es desactivado. Si miramos el valor de máscara `0022`, podemos ver lo que hace:

Modo de archivo original	`--- rw- rw- rw-`
Máscara	`000 000 010 010`
Resultado	`--- rw- r-- r--`

De nuevo, donde aparece un 1 en el valor binario, el correspondiente atributo es desactivado. Prueba con algunos valores (prueba algunos sietes) para acostumbrarte a su funcionamiento. Cuando lo hayas hecho, recuerda limpiar:

```
[me@linuxbox ~]$ rm foo.txt; umask 0002
```

La mayoría de las veces no tendrás que cambiar la máscara; la que tu distribución trae por defecto irá bien. En algunas situaciones de alta seguridad, sin embargo, querrás controlarla.

Algunos permisos especiales

Aunque a veces vemos la máscara octal de permisos expresada como un número de tres dígitos, es más correcto técnicamente expresarla en cuatro dígitos ¿por qué? Porque, además de permisos de lectura, escritura y ejecución, hay algunas otras configuraciones de permisos menos usadas.

El primero de ellos es el *setuid bit* (octal 4000). Cuando lo aplicamos a un archivo ejecutable, cambia la *ID efectiva de usuario* del usuario real (el usuario que realmente está utilizando el programa) a la del propietario del programa. Muy a menudo esto se aplica a unos pocos programas

cuyo propietario es el superusuario. Cuando un usuario ordinario ejecuta un programa que está "*setuid root*", el programa funciona con los privilegios efectivos del superusuario. Esto permite al programa acceder a archivos y directorios a los que un usuario ordinario tendría normalmente prohibido su acceso. Por supuesto, debido que esto plantea problemas de seguridad, el número de programas setuid debe reducirse al mínimo necesario.

El segundo permiso menos usado es el *setgid bit* (octal 2000) el cual, como el setuid bit, cambia el *ID efectivo de grupo* de el *ID real de grupo* al del propietario del archivo. Si el setgid bit se aplica en un directorio, los nuevos archivos que se creen en el directorio tendrán como grupo del propietario el del directorio en lugar del de creador del archivo. Esto es útil en un directorio compartido cuando los miembros de un grupo común necesitan acceso a todos los archivos del directorio, independientemente del grupo del propietario primario del archivo.

El tercero se llama el *sticky bit* (octal 1000). Esto es una reliquia del antiguo Unix, donde era posible marcar un archivo ejecutable como "no intercambiable". En archivos, Linux ignora el sticky bit, pero si se aplica a un directorio, previene que los usuarios borren o renombren archivos a menos que el usuario sea el propietario del directorio, el propietario del archivo o el superusuario. Esto, a menudo, se usa para controlar el acceso a un directorio compartido, como `/tmp`.

Aquí hay algunos ejemplos de uso de `chmod` con notación simbólica para establecer estos permisos especiales. Primero asignaremos setuid a un programa:

```
chmod u+s program
```

A continuación, asignaremos setgid a un directorio:

```
chmod g+s dir
```

Finalmente, asignaremos el sticky bit a un directorio:

```
chmod +t dir
```

Cuando vemos la salida de ls, podemos determinar los permisos especiales. Aquí tenemos algunos ejemplos, primero un programa que tiene setuid:

```
-rwsr-xr-x
```

Un directorio que tiene el atributo setgid:

```
drwxrwsr-x
```

Un directorio con el sticky bit establecido:

```
drwxrwxrwt
```

Cambiar identidades

Muchas veces, encontraremos necesario tomar la identidad de otro usuario. A menudo querremos ganar permisos de superusuario para manejar tareas administrativas, pero también es posible "convertirse" en otro usuario normal para cosas como probar una cuenta. Hay tres formas para tomar una identidad alternativa:

1. Cerrar la sesión y volver a abrirla como el usuario alternativo.
2. Usar el comando `su`.
3. Usar el comando `sudo`.

Nos saltaremos la primera técnica ya que sabemos como hacerlo y además no tiene las ventajas de las otras dos. Desde dentro de nuestra propia sesión de shell, el comando su nos permite asumir la identidad de otro usuario, abriendo una nueva sesión de shell con la ID de ese usuario o ejecutar un comando como ese usuario. El comando `sudo` permite a un administrador modificar un archivo de configuración llamado `/etc/sudoers`, y definir comandos específicos que usuarios particulares tienen permiso ejecutar bajo un identidad asumida. La elección de qué comando usar está muy condicionada por la distribución Linux que uses. Tu distribución probablemente incluya los dos comandos, pero su configuración favorecerá un comando u otro. Empezaremos con `su`.

su – Ejecutar un Shell con Ids sustitutos de usuarios y grupos

El comando `su` se usar para arrancar un shell como otro usuario. La sintaxis del comando es así:

```
su [-[l]] [user]
```

Si se incluye la opción "-l", la sesión shell resultante es un *shell* de login para el usuario especificado. Ésto significa que el entorno del usuario se carga y el directorio de trabajo se cambia al directorio home del usuario. Ésto es lo que queremos normalmente. Si no especificamos el usuario, se asume el superusuario. Fíjate que (extrañamente) puede ser abreviada como "-", que es como se usa más a menudo. Para arrancar una shell para el superusuario, haríamos esto:

```
[me@linuxbox ~]$ su -Password:[root@linuxbox ~]#
```

Después de introducir el comando, nos pide la contraseña del superusuario. Si la introducimos correctamente, un nuevo prompt de shell aparece indicando que ésta nueva shell tiene privilegios de superusuario ("#" en lugar de "$") y el directorio de trabajo actual es ahora el directorio home del superusuario (normalmente `/root`.). Una vez que estamos en el nuevo shell, podemos ejecutar comandos como superusuario. Cuando terminemos, escribimos "exit" para volver al shell previo:

```
[root@linuxbox ~]# exit[me@linuxbox ~]$
```

También es posible ejecutar un único comando en lugar de comenzar un nuevo comando interactivo usando su de la siguiente forma:

```
su -c 'comando'
```

Usando esta forma, un única línea de comandos es pasada al nuevo shell para su ejecución. Es importante incluir el comando entre comillas simples, ya que no queremos que se realice una expansión en nuestro shell, ni tampoco en el nuevo shell:

```
[me@linuxbox ~]$ su -c 'ls -l /root/*'Password:-rw------- 1
root root 754 2007-08-11 03:19 /root/anaconda-
ks.cfg/root/Mail:total 0[me@linuxbox ~]$
```

sudo – Ejecutar un comando como otro usuario

El comando `sudo` es como su en muchos aspectos, pero tiene algunas capacidades adicionales importantes. El administrador puede configurar `sudo` para permitir a los usuarios ordinarios ejecutar comandos como un usuario diferente (normalmente el superusuario) de una forma muy controlada. En particular, un usuario estaría limitado a uno o más comandos específicos y no a otros. Otra diferencia importante es que el uso de `sudo` no requiere la contraseña del superusuario. Para autenticarse usando `sudo`, el usuario utiliza su propia contraseña. Digamos, por ejemplo, que `sudo` ha sido configurado para permitirnos ejecutar una programa ficticio de copias de seguridad

llamado “backup_script”, que requiere privilegios de superusuario. Con sudo podría hacerse así:

```
[me@linuxbox ~]$ sudo backup_scriptPassword:System Backup
Starting...
```

Después de introducir el comando, somo preguntados por nuestra contraseña (no la del superusuario) y una vez que la autenticación está completada, el comando especificado se lleva a cabo. Una diferencia importante entre su y sudo es que sudo no abre una nueva shell, ni carga el entorno de otro usuario. Esto significa que los comandos no necesitan ser entrecomillados de forma diferente a la que lo estarían si lo ejecutáramos sin usar sudo. Ten en cuenta que este comportamiento puede ser anulado especificando varias opciones. Mira la man page de sudo para más detalles.

Para ver qué privilegios están aceptados por sudo, usa la opción “-l” para listarlos:

```
[me@linuxbox ~]$ sudo -lUser me may run the following
commands on this host:(ALL) ALL
```

Ubuntu y sudo

Uno de los problemas recurrentes para los usuarios normales es ejecutar ciertas tareas que requieren privilegios de superusuario. Estas tareas incluyen instalar y actualizar software, editar archivos de configuración del sistema, y acceder a dispositivos. En el mundo Windows, se hace a menudo dando a los usuarios privilegios de administrador. Ésto permite a los usuarios ejecutar estas tareas. Sin embargo, también permite que los programas ejecutados por el usuario tengan las mismas capacidades. Esto no es deseable en la mayoría de los casos, ya que permite al *malware* (sortware malicioso) como virus tener dominio absoluto del ordenador. En el mundo Unix, siempre ha habido una gran división entre los usuarios normales y los administradores, debido a la herencia multiusuario de Unix. El enfoque tomado en Unix es proporcionar privilegios de superusuario sólo cuando se necesite. Para hacer ésto, se usan comúnmente los comando su y sudo.

Hasta hace pocos años, la mayoría de las distribuciones Linux confiaban en su para éste propósito. su no requiere la configuración que requiere sudo, y tener una cuenta root es tradicional en Unix. Esto creaba un problema. Los usuarios tendían a operar como root cuando no era necesario. De hecho, algunos usuarios operaban sus sistemas como root exclusivamente, ya que eliminaban todos esos molestos mensajes de “permiso denegado”. Así es como se reduce la seguridad de un sistema Linux a la de un sistema Windows. No es una buena idea.

Cuando apareció Ubuntu, sus creadores tomaron un rumbo diferente. Por defecto, Ubuntu desactiva el acceso a la cuenta root (impidiendo establecer una contraseña para la cuenta), y en lugar utiliza sudo para proporcionar privilegios de superusuario. La cuenta de usuario inicial tiene acceso a privilegios completos de superusuario vía sudo y pueden proporcionar poderes similares a posteriores cuentas de usuario.

chown – Cambia el propietario y el grupo de un archivo

El comando chown se utiliza para cambiar el propietario y el grupo del propietario de un archivo o un directorio. Se requieren privilegios de superusuario para utilizar este comando. La sintaxis de chown tiene este aspecto:

```
chown [propietario][:[grupo]] archivo...
```

chown puede cambiar el propietario del archivo y/o el grupo del propietario dependiendo del primer argumento del comando. Aquí tenemos algunos ejemplos:

Tabla 9-7: Ejemplos de argumentos de chown

Argumento	Resultado
bob	Cambia el propietario del archivo del propietario actual al usuario bob.
bob:users	Cambia la propiedad del archivo de su actual propietario al usuario bob y cambia el grupo del propietario del archivo al grupo users.
:admins	Cambia el grupo del propietario al grupo admins. El propietario del archivo no cambia
bob:	Cambia el propietario del archivo del propietario actual al usuario bob y cambia el grupo del propietario al grupo de acceso del usuario bob.

Digamos que tenemos dos usuarios; janet, quien tiene acceso a privilegios de superusuario y tony, que no los tiene. El usuario janet quiere copiar un archivo de su directorio home al directorio home del usuario tony. Como el usuario janet quiere que tony pueda editar el archivo, janet cambia la propiedad del archivo copiado de janet a tony:

```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tonyPassword:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt-rw-r--r-- 1
root root 8031 2008-03-20 14:30
/home/tony/myfile.txt[janet@linuxbox ~]$ sudo chown tony:
~tony/myfile.txt[janet@linuxbox ~]$ sudo ls -l
~tony/myfile.txt-rw-r--r-- 1 tony tony 8031 2008-03-20
14:30 /home/tony/myfile.txt
```

Aquí vemos que el usuario janet copia el archivo de su directorio al directorio home del usuario tony. A continuación, janet cambia la propiedad del archivo de root (como resultado de usar sudo) a tony. Usando los dos puntos en el primer argumento, janet también ha cambiado el grupo del propietario del archivo al grupo de acceso de tony, que resulta ser el grupo tony.

Fíjate que tras el primer uso de sudo, janet ¿no es preguntada por su usuario? Ésto es porque sudo, en la mayoría de las configuraciones, "confía" en ti unos minutos antes de que su temporizador se cierra.

chgrp – Cambiando el grupo del propietario

En antiguas versiones de Unix, el comando chown sólo cambiaba la propiedad del archivo, no el grupo del propietario. Para ese propósito, un comando a parte, chgrp era el utilizado. Funciona de forma muy parecida a chown, excepto en que es mucho más limitado.

Ejercitando nuestros privilegios

Ahora que hemos aprendido como funciona esto de los permisos, es hora de lucirse. Vamos a demostrar la solución a un problema común – configurar un directorio compartido. Imaginemos que tenemos dos usuarios llamados "bill" y "karen". Ambos tiene colecciones de CDs de música y les gustaría crear un directorio compartido, donde cada uno guarde sus archivos de música como Ogg Vorbis o MP3. El usuario bill tiene acceso a privilegios de superusuario vía sudo.

La primera cosa que tiene que ocurrir es crear un grupo que tenga a bill y a karen como

miembros. Usando la herramienta gráfica de gestión de usuarios, `bill` crea un grupo llamado `music` y le añade los usuarios `bill` y `karen`:

Figura 3: Creando un nuevo grupo con GNOME

A continuación, `bill` crea el directorio para los archivos de música:

```
[bill@linuxbox ~]$ sudo mkdir
/usr/local/share/MusicPassword:
```

Como `bill` está manipulando archivos fuera de su directorio home, requiere privilegios de superusuario. Después de crear el directorio, tendrá los siguientes propietarios y permisos:

```
[bill@linuxbox ~]$ ls -ld /usr/local/share/Musicdrwxr-xr-x
2 root root 4096 2008-03-21 18:05 /usr/local/share/Music
```

Como podemos ver, el directorio tiene como propietarios al `root` y tiene permisos 755. Para hacer este directorio compartible, `bill` necesita cambiar el grupo del propietario y sus permisos para permitir la escritura:

```
[bill@linuxbox ~]$ sudo chown :music
/usr/local/share/Music[bill@linuxbox ~]$ sudo chmod 775
/usr/local/share/Music[bill@linuxbox ~]$ ls -ld
/usr/local/share/Musicdrwxrwxr-x 2 root music 4096 2008-03-
21 18:05 /usr/local/share/Music
```

¿Pero qué significa todo ésto? Significa que ahora tenemos un directorio, `/usr/local/share/Music` cuyo propietario es `root` y permite acceso de lectura y escritura al grupo `music`. El grupo `music` tiene como miembros al `bill` y `karen`, tanto `bill` como `karen` pueden crear archivos en el directorio `/usr/local/share/Music`. Otros usuarios pueden listar el contenido del directorio pero no pueden crear archivos dentro.

Pero aún tenemos un problema. Con los permisos actuales, los archivos y directorios creados en el directorio Music tendrán los permisos normales de los usuarios `bill` y `karen`:

```
[bill@linuxbox ~]$ >
/usr/local/share/Music/test_file[bill@linuxbox ~]$ ls -l
/usr/local/share/Music-rw-r--r-- 1 bill bill 0 2008-03-24
20:03 test_file
```

Realmente tenemos dos problemas. Primero, la `umask` en este sistema es 0022 la cual previene que

los miembros del grupo no puedan escribir archivos pertenecientes a otros miembros del grupo. Ésto no sería un problema si el directorio compartido sólo contuviera archivos, pero como este directorio contendrá música, y la música suele organizarse en una jerarquía de artistas y álbumes, los miembros del grupo necesitarán la capacidad de crear archivos y directorios dentro de los directorios creados por otros miembros. Necesitamos modificar la `umask` utilizada por `bill` y `karen` a 0002.

Segundo, cada archivo y directorio creado por un miembro será adjudicado al grupo primario del usuario en lugar de al grupo música. Ésto se puede arreglar estableciendo el setgid bit en el directorio:

```
[bill@linuxbox ~]$ sudo chmod g+s
/usr/local/share/Music[bill@linuxbox ~]$ ls -ld
/usr/local/share/Musicdrwxrwsr-x 2 root music 4096 2008-03-
24 20:03 /usr/local/share/Music
```

Ahora probaremos a ver si los nuevos permisos arreglan el problema. `bill` establece su `umask` a 0002, elimina el anterior archivo de prueba, y crea un nuevo archivo y directorio de pruebas:

```
[bill@linuxbox ~]$ umask 0002[bill@linuxbox ~]$ rm
/usr/local/share/Music/test_file[bill@linuxbox ~]$ >
/usr/local/share/Music/test_file[bill@linuxbox ~]$ mkdir
/usr/local/share/Music/test_dir[bill@linuxbox ~]$ ls -l
/usr/local/share/Musicdrwxrwsr-x 2 bill music 4096 2008-03-
24 20:24 test_dir-rw-rw-r-- 1 bill music 0 2008-03-24 20:22
test_file[bill@linuxbox ~]$
```

Tanto archivos como directorios se crean ahora con los permisos correctos para permitir a todos los miembros del grupo `music` crear archivos y directorios dentro del directorio `Music`.

El único problema que queda es `umask`. La configuración necesaria sólo dura hasta el final de la sesión y debe ser reconfigurada. En el Capítulo 11, veremos como hacer el cambio de umask permanente.

Cambiando tu contraseña

El último asunto que veremos en este capítulo es establecer contraseñas para tí mismo (y para otros usuarios si tienes acceso a privilegios de superusuario). Para establecer o cambiar una contraseña, se usa el comando passwd. La sintaxis del comando es así:

```
passwd [usuario]
```

Para cambiar tu contraseña, sólo introduce el comando `passwd`. Serás preguntado por tu contraseña anterior y por la nueva:

```
[me@linuxbox ~]$ passwd(current) UNIX password:New UNIX
password:
```

El comando `passwd` tratará de obligarte a usar contraseñas "fuertes". Ésto significa que rechazará contraseñas que son muy cortas, muy parecidas a contraseñas anteriores, palabras del diccionario, o muy fácilmente adivinables:

```
[me@linuxbox ~]$ passwd(current) UNIX password:New UNIX
password:BAD PASSWORD: is too similar to the old oneNew
```

```
UNIX password:BAD PASSWORD: it is WAY too shortNew UNIX
password:BAD PASSWORD: it is based on a dictionary word
```

Si tienes privilegios de superusuario, puedes especificar un nombre de usuario como argumento en el comando `passwd` para establecer la contraseña de otro usuario. Hay otras opciones disponibles para el superusuario como, el bloqueo de cuentas, caducidad de la contraseña, etc. Mira la man page de `passwd` para más detalles.

Resumiendo

En este capítulo hemos visto como los sistemas como-Unix, por ejemplo Linux manejan los permisos de usuario para permitirles acceso para leer, escribir y ejecutar archivos y directorios. La idea básica de este sistema de permisos viene de los primeros días de Unix y se han mantenido muy bien con el paso del tiempo. Pero el mecanismo nativo de permisos en los sistemas Unix-like carecen de la granularidad precisa de sistemas más modernos.

Para saber más

- Wikipedia tiene un buen artículo sobre malware:

  http://en.wikipedia.org/wiki/Malware

Hay numerosos programas de línea de comandos para crear y mantener usuarios y grupos. Para más información, mira las man pages de los siguientes comandos:

- `adduser`
- `useradd`
- `groupadd`

Procesos

Los sistemas operativos modernos son normalmente *multitarea*, lo que significa que crean la ilusión de que hacen más de una cosa al mismo tiempo mediante un rápido cambio de un programa en ejecución a otro. El kernel Linux gestiona ésto a través de el uso de *procesos*. Los procesos son la forma en que Linux organiza los diferentes programas que esperan su turno en la CPU.

Algunas veces un ordenador se vuelve lento o una aplicación deja de responder. En este capítulo, veremos algunas de las herramientas disponibles en la línea de comandos que nos permitirá examinar qué están haciendo los programas, y como terminar procesos que están teniendo un mal comportamiento.

Éste capítulo presentará los siguientes comandos:

- `ps` – Muestra un listado de los procesos en funcionamiento
- `top` – Muestra las tareas
- `jobs` – Lista los trabajos activos
- `bg` – Coloca un trabajo al fondo (background)
- `fg` – Coloca un trabajo al frente (foreground)
- `kill` – Manda un señal a un proceso
- `killall` – Mata procesos por nombre
- `shutdown` – Apaga o reinicia el sistema

Cómo funciona un proceso

Cuando un sistema arranca, el kernel inicia algunas de sus propias actividades como procesos y arranca un programa llamado `init`. `init`, por orden, ejecuta una serie de scripts de shell (localizados en `/etc`) llamados *init scripts*, que arrancan todos los servicios del sistema. Muchos de estos servicios son implementados como *daemon programs* (demonios), programas que simplemente se sitúan en el fondo y hacen su trabajo sin tener ningún usuario en la interface. Así que incluso sin que estemos logueados, el sistema está un poco ocupado haciendo tareas rutinarias.

El hecho de que un programa pueda arrancar otros programas se expresa en el esquema de procesos como *procesos padres* produciendo *procesos hijos*.

El kernel mantiene información sobre cada proceso para ayudar a tener las cosas organizadas. Por ejemplo, cada proceso tiene asignado un número llamado *`process ID`* o *`PID`*. Los PIDs son asignados en orden creciente, con `init` siempre ocupando el PID1. El kernel también guarda la ruta a la memoria asignada a cada proceso, así como la disponibilidad de los procesos para reanudar la ejecución. Como los archivos, los procesos también tiene propietarios y IDs de usuario, IDs efectivas de usuarios, etc.

Viendo los procesos

El comando más comúnmente usado para ver procesos (hay varios) es `ps`. El programa `ps` tiene un montón de opciones, pero en su forma más simple se usa así:

```
[me@linuxbox ~]$ ps
PID TTY TIME CMD
5198 pts/1 00:00:00 bash
10129 pts/1 00:00:00 ps
```

El resultado de este ejemplo es un lista de dos procesos, el proceso 5198 y el proceso 10129, que son `bash` y `ps` respectivamente. Como podemos ver, por defecto, `ps` no nos muestra mucho, sólo los procesos asociados a la sesión de terminal actual. Para ver más, necesitamos añadir algunas opciones, pero antes de hacerlo, veamos los otros campos producidos por `ps`. `TTY` es la abreviatura de "Teletype", y se refiere al *terminal que controla* el proceso. Unix muestra su edad aquí. El comando `TIME` es la cantidad de tiempo consumido por el proceso. Como podemos ver, ninguno de los procesos hace al ordenador trabajar mucho.

Si añadimos una opción, podemos tener una imagen más amplia de que está haciendo el sistema:

```
[me@linuxbox ~]$ ps x
PID   TTY STAT TIME COMMAND
2799  ?   Ssl  0:00 /usr/libexec/bonobo-activation-server –ac
2820  ?   Sl   0:01 /usr/libexec/evolution-data-server-1.10 --
15647 ?   Ss   0:00 /bin/sh /usr/bin/startkde
15751 ?   Ss   0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --
15754 ?   S    0:00 /usr/bin/dbus-launch --exit-with-session
15755 ?   Ss   0:01 /bin/dbus-daemon --fork --print-pid 4 –pr
15774 ?   Ss   0:02 /usr/bin/gpg-agent -s –daemon
15793 ?   S    0:00 start_kdeinit --new-startup +kcminit_start
15794 ?   Ss   0:00 kdeinit Running...
15797 ?   S    0:00 dcopserver –nosid
```

```
and many more...
```

Añadiendo la opción “x” (fíjate que no lleva un guión delante) le decimos a `ps` que muestre todos nuestros procesos independientemente de qué terminal (si hay alguno) los controla. La presencia de un “?” en la columna `TTY` indica que ningún terminal controla. Usando esta opción, vemos una lista de cada proceso que tenemos.

Como el sistema está ejecutando muchos procesos, `ps` produce una lista larga. A menudo ayuda pasar la salida por un pipe `ps` a `less` para verlo más fácilmente. Algunas combinaciones de opciones también producen salidas con listas largas, así que maximizar la ventana del emulador de terminal puede ser una buena idea también.

Una nueva columna llamada `STAT` se ha añadido a la salida. `STAT` es una abreviatura de “state” y revela el estado actual del proceso:

Tabla 10-1: Estado de los procesos

Estado	Significado
`R`	Funcionando (running). Significa que el proceso está en ejecución o preparado para ser ejecutado.
`S`	Durmiendo (Sleeping). El proceso no está ejecutándose; en su lugar, está esperando a un evento, como una pulsación de teclado o un paquete de red.
`D`	Ininterrumpible dormido. El proceso está esperando un I/O como una unidad de disco.
`T`	Parado. El proceso ha sido ordenado a parar. Veremos más sobre esto más tarde.
`Z`	Proceso extinto o “zombie”. Es un proceso hijo que ha terminado, pero no ha sido limpiado por su padre.
`<`	Un proceso de alta prioridad. Es posible dar más importancia a un proceso, dándole más tiempo en la CPU. Esta propiedad de un proceso se llama *simpatía*. Un proceso con alta prioridad se dice que es menos *simpático* porque esta utilizando más tiempo de la CPU, lo que deja menos tiempo para todos los demás.
`N`	Un proceso de baja prioridad. Un proceso con baja prioridad (un proceso “amable”) sólo ocupará tiempo de procesador después de que otros procesos con mayor prioridad hayan sido servidos.

El estado del proceso puede ir seguido de otros caracteres. Ésto indica varias características exóticas de los procesos. Mira la man page de `ps` para más detalles.

Otro popular grupo de opciones es “aux” (sin guión delante). Nos da incluso más información:

```
[me@linuxbox ~]$ ps aux
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1   0.0  0.0  2136 644  ?   Ss Mar05 0:31 init
root 2   0.0  0.0     0   0  ?   S< Mar05 0:00 [kt]
root 3   0.0  0.0     0   0  ?   S< Mar05 0:00 [mi]
root 4   0.0  0.0     0   0  ?   S< Mar05 0:00 [ks]
root 5   0.0  0.0     0   0  ?   S< Mar05 0:06 [wa]
root 6   0.0  0.0     0   0  ?   S< Mar05 0:36 [ev]
root 7   0.0  0.0     0   0  ?   S< Mar05 0:00 [kh]
and many more...
```

Esta configuración de opciones muestras los procesos pertenecientes a cada usuario. Usando las opciones sin el guión al principio invoca el comando con comportamiento "estilo BSD". La versión Linux de ps puede emular el comportamiento del programa ps que se encuentra en varias implementaciones de Unix. Con estas opciones, tendemos estas columnas adicionales:

Tabla 10-2: Encabezados de Columnas en el Estilo BSD de ps

Encabezado	Significado
USER	ID de usuario. Es el propietario del proceso.
%CPU	Uso de CPU en porcentaje.
%MEM	Uso de memoria en porcentaje.
VSZ	Tamaño de la memoria virtual.
RSS	Tamaño de configuración residente. La cantidad de memoria física (RAM) que el proceso usa en kilobytes.
START	Hora en que comenzó el proceso. Para valores de más de 24 horas, se usa la fecha.

Viendo los procesos dinámicamente con top

Mientras que el comando ps puede revelar mucho de lo que está haciendo la máquina, sólo proporciona una fotografía del estado de la máquina en el momento en que se ejecuta el comando ps. Para ver una vista más dinámica de la actividad de la máquina, usamos el comando top:

```
[me@linuxbox ~]$ top
```

El programa top muestra una pantalla en actualización constante (por defecto, cada 3 segundos) con los procesos del sistema listados por orden de actividad de los procesos. El nombre "top" viene del hecho de que el programa top se usa para ver los procesos "top" del sistema. La pantalla de top contiene dos partes: un sumario del sistema en la parte de arriba de la pantalla, seguido de una tabla de los procesos ordenados por actividad de la CPU:

```
top - 14:59:20 up 6:30, 2 users, load average: 0.07, 0.02, 0.00
Tasks: 109 total, 1 running, 106 sleeping, 0 stopped, 2 zombie
Cpu(s): 0.7%us, 1.0%sy, 0.0%ni, 98.3%id, 0.0%wa, 0.0%hi, 0.0%si
Mem: 319496k total, 314860k used, 4636k free, 19392k buff
Swap: 875500k total, 149128k used, 726372k free, 114676k cach
PID   USER PR NI VIRT  RES  SHR  S %CPU %MEM TIME+    COMMAND
6244  me   39 19 31752 3124 2188 S 6.3  1.0  16:24.42 trackerd
11071 me   20  0  2304 1092  840 R 1.3  0.3   0:00.14 top
6180  me   20  0  2700 1100  772 S 0.7  0.3   0:03.66 dbus-dae
6321  me   20  0 20944 7248 6560 S 0.7  2.3   2:51.38 multiloa
4955  root 20  0  104m 9668 5776 S 0.3  3.0   2:19.39 Xorg
1     root 20  0  2976  528  476 S 0.0  0.2   0:03.14 init
2     root 15 -5     0    0    0 S 0.0  0.0   0:00.00 kthreadd
3     root RT -5     0    0    0 S 0.0  0.0   0:00.00 migratio
4     root 15 -5     0    0    0 S 0.0  0.0   0:00.72 ksoftirq
5     root RT -5     0    0    0 S 0.0  0.0   0:00.04 watchdog
6     root 15 -5     0    0    0 S 0.0  0.0   0:00.42 events/0
7     root 15 -5     0    0    0 S 0.0  0.0   0:00.06 khelper
41    root 15 -5     0    0    0 S 0.0  0.0   0:01.08 kblockd/
67    root 15 -5     0    0    0 S 0.0  0.0   0:00.00 kseriod
```

```
114   root 20  0     0    0    0 S 0.0  0.0   0:01.62 pdflush
116   root 15 -5     0    0    0 S 0.0  0.0   0:02.44 kswapd0
```

El sumario del sistema contiene mucho material bueno. Aquí tenemos un repaso:

Tabla 10-3: Campos de información de top

Columna	Campo	Significado
1	`top`	Nombre del programa
	`14:59:20`	Hora actual del día.
	`up 6:30`	Ésto se llama *uptime*. Es la cantidad de tiempo desde que la máquina fue arrancada. En este ejemplo, el sistema ha estado funcionando seis horas y media.
	`2 users`	Hay dos usuarios logueados.
	`Load average:`	*Load average* se refiere al número de procesos que están esperando para funcionar, o sea, el número de procesos que están en un estado ejecutable y están compartiendo la CPU. Se muestran tres valores, cada uno para un periodo de tiempo diferente. El primero es la media de los últimos 60 segundos, el siguiente los 5 minutos anteriores, y finalmente los últimos 15 minutos. Los valores menores de 1.0 indican que la máquina no está ocupada
2	`Tasks:`	Resume el número de procesos y sus distintos estados de proceso.
3	`Cpu(s):`	Esta columna describe el carácter de las actividades que la CPU está realizando.
	`0.7%us`	El 0.7% de la CPU está siendo usada por *procesos de usuario*. Ésto significa que son procesos fuera del propio kernel.
	`1.0%sy`	El 1.0% de la CPU está siendo usada por procesos “*del sistema* (kernel)”
	`0.0%ni`	El 0.0% de la CPU está siendo usada por procesos “simpáticos” (de baja prioridad).
	`98.3%id`	El 98.3% de la CPU está desocupada.
	`0.0%wa`	0.0% de la CPU está esperado I/O.
4	`Mem:`	Muestra como está siendo usada la memoria física RAM.
5	`Swap:`	Muestra cómo está siendo usado el espacio de intercambio (memoria virtual).

El programa top acepta comandos de teclado. Los dos más interesantes son `h`, que muestra la pantalla de ayuda del programa, y `q`, que cierra `top`.

Los dos principales entornos de escritorio proveen aplicaciones gráficas que muestran información similar a `top` (de la misma forma que lo hace el Administrador de Tareas de Windows), pero encuentro que `top` es mejor que las versiones gráficas porque es más rápido y consume muchos menos recursos del sistema. Después de todo, nuestro programa monitor del sistema no debería ser la causa de la lentitud del sistema que estamos intentando encontrar.

Controlando procesos

Ahora que podemos ver y monitorizar procesos, consigamos controlarlos. Para nuestros experimentos, vamos a utilizar un pequeño programa llamado `xlogo` como conejillo de indias. El programa `xlogo` es un ejemplo de programa proporcionado por el sistema X Window (el motor en segundo plano que hace que los gráficos se vean en nuestra pantalla) que simplemente muestra una ventana modificable en tamaño que contiene el logo de X. Primero, vamos a conocer a nuestro sujeto de pruebas:

```
[me@linuxbox ~]$ xlogo
```

Después de introducir el comando, una pequeña ventana conteniendo el logo debería aparecer en algún sitio de la pantalla. En nuestros sistemas, `xlogo` debería mostrar un mensaje de advertencia, pero puede ser ignorado tranquilamente.

Consejo: Si tu sistema no incluye el programa `xlogo`, preba a usar `gedit` o `kwrite` en su lugar.

Podemos verificar que `xlogo` está funcionando modificando el tamaño de la ventana. Si el logo se redibuja en el nuevo tamaño, el programa está funcionando.

¿Ves que nuestro prompt de shell no ha vuelto? Esto es porque el shell está esperando a que el programa termine, como todos los demás programas que hemos usado hasta ahora. Si cerramos la ventana de `xlogo`, el prompt vuelve.

Interrumpiendo un proceso

Observemos que ocurre cuando ejecutamos el programa `xlogo` de nuevo. Primero, introducimos el comando `xlogo` y verificamos que el programa está funcionando. A continuación, volvemos a la ventana del terminal y pulsamos `Ctrl-c`.

```
[me@linuxbox ~]$ xlogo[me@linuxbox ~]$
```

En un terminal, pulsar `Ctrl-c`, *interrumpe* un programa. Esto significa que le pedimos educadamente al programa que termine. Después de pulsar `Ctrl-c`, la ventana de `xlogo` se cierra y el prompt de shell vuelve.

Muchos (pero no todos) programas de la línea de comandos pueden ser interrumpidos usando esta técnica.

Enviando procesos al fondo

Digamos que queremos traer de vuelta el prompt de shell sin terminar el programa `xlogo.` Lo haremos enviando el programa al *fondo*. Piensa en el terminal como si tuviera un *frente* (con cosas visibles en la superficie como el prompt del shell) y un fondo (con cosas ocultas bajo la superficie). Para arrancar un programa y que se coloque inmediatamente en el fondo, añadiremos al comando un carácter “&”:

```
[me@linuxbox ~]$ xlogo &[1] 28236[me@linuxbox ~]$
```

Después de introducir el comando, aparece la ventana de `xlogo` y el prompt de shell vuelve, pero algunos números extraños también aparecen. Este mensaje es parte de una característica del shell llamada *control de trabajo*. Con este mensaje, el shell nos está diciendo que hemos comenzado el

trabajo número 1 ("[1]") y que tiene el PID 28236. Si ejecutamos `ps`, podemos ver nuestro proceso:

```
[me@linuxbox ~]$ psPID TTY TIME CMD10603 pts/1 00:00:00
bash28236 pts/1 00:00:00 xlogo28239 pts/1 00:00:00 ps
```

La utilidad de control de trabajos del shell también nos da una forma de listar los trabajos que hemos arrancado desde nuestro terminal. Utilizando el comando `jobs`, podemos ver esta lista:

```
[me@linuxbox ~]$ jobs[1]+ Running xlogo &
```

Los resultados muestran que tenemos un trabajo, con el número "1", y que esta funcionando, y que el comando fue `xlogo &`.

Devolviendo un proceso al frente

Un proceso en el fondo es inmune a nuestras entradas de teclado, incluyendo cualquier intento de interrumpirlo con `Ctrl-c`. Para devolver un proceso al frente, usa el comando `fg`, de esta forma:

```
[me@linuxbox ~]$ jobs[1]+ Running xlogo &[me@linuxbox ~]$
fg %1xlogo
```

El comando `fg` seguido de un signo de porcentaje y el número del trabajo (llamado *jobspec*) hace el truco. Si sólo tenemos un trabajo en el fondo, el jobspec es opcional. Para terminar `xlogo`, presiona `Ctrl-c`.

Parando (pausando) un proceso

A veces queremos parar un proceso sin terminarlo. Esto se hace a menudo para permitir que un proceso que está en el frente sea movido al fondo. Para parar un proceso que está en el frente, pulsa `Ctrl-z`. Pruébalo. En el prompt del comando, escribe `xlogo`, pulsa `ENTER` y luego `Ctrl-z`:

```
[me@linuxbox ~]$ xlogo[1]+ Stopped xlogo[me@linuxbox ~]$
```

Después de parar `xlogo`, podemos verificar que el programa se ha detenido intentando redimensionar la ventana de `xlogo`. Veremos que parece un poco muerta. Podemos también devolver el programa al frente, usando el comando `fg`, o mover el programa al fondo con el comando `bg`:

```
[me@linuxbox ~]$ bg %1[1]+ xlogo &[me@linuxbox ~]$
```

Como en el comando `fg`, el jobspec es opcional si sólo hay un trabajo.

Mover un proceso del frente al fondo es útil si arrancamos un programa gráfico desde comandos, pero olvidas colocarlo en el fondo añadiendo el "&" delante.

¿Por qué querrías arrancar un programa gráfico desde la línea de comandos? Hay dos razones. Primero, el programa que quieres iniciar no está incluido en los menús del gestor de ventanas (como `xlogo`). En segundo lugar, ejecutando un programa desde la línea de comando, podrías ver mensajes de error que sería invisibles si fuera ejecutado gráficamente. A veces, un programa puede fallar si se inicia desde el menú gráfico. Arrancándolo desde la línea de comandos verás un mensaje de error que revelará el problema. Además, algunos programas gráficos pueden tener muchas

opciones interesantes y útiles en la línea de comandos.

Señales

El comando `kill` se usa para *matar* procesos. Esto nos permite terminar programas que necesitan ser matados. Aquí tenemos un ejemplo:

```
[me@linuxbox ~]$ xlogo &[1] 28401[me@linuxbox ~]$ kill
28401[1]+ Terminated xlogo
```

Primero arrancamos `xlogo` en el fondo. El shell muestra el jobspec y el PID del proceso en el fondo. A continuación, usamos el comando `kill` y especificamos el PID del proceso que queremos terminar. Podríamos haber especificado también el proceso usando un jobspec (por ejemplo, “%1”) en lugar de un PID.

Aunque que esto es todo muy sencillo, hay mucho más al respecto. El comando `kill` no mata exactamente los procesos, en su lugar les manda *señales*. Las señales son uno de los distintos caminos con que el sistema operativo se comunica con los programas. Ya hemos visto señales en acción con el uso de `Ctrl-c` y `Ctrl-z`. Cuando un terminal recibe una de estas pulsaciones de teclado, manda una señal al programa en el frente. En el caso de `Ctrl-c`, se manda una señal llamada `INT` (Interrumpción); con `Ctrl-z`, una señal llamada `TSTP` (Terminal Stop). Los programas, por turnos, “escuchan” las señales y las utilizan según las van recibiendo. El hecho de que un programa pueda escuchar y actuar mediante señales permite que el programa haga cosas como salvar el trabajo en proceso cuando se le envía una seña de terminación.

Enviando señales a un proceso con kill

El comando `kill` se usa para enviar señales a programas. Su sintaxis más común es así:

```
kill [-signal] PID...
```

Si no se especifican señales en la línea de comandos, la señal `TERM` (Terminar) se envía por defecto. El comando `kill` se usa la mayoría de las veces para enviar las siguientes señales:

Tabla 10-4: Señales comunes

Número	Nombre	Significado
1	`HUP`	Colgar. Es un vestigio de los buenos y viejos tiempos cuando los terminales estaban conectados a ordenadores remotos mediante líneas telefónicas y modems. La señal se usa para indicar a los programas que el terminal que controla ha “colgado”. El efecto de esta señal puede demostrarse cerrando una sesión de terminal. El programa en el frente que está corriendo en el terminal recibirá una señal y terminará. Esta señal también se usa por muchos programas demonio para provocar una reinicialización. Esto significa que cuando se le envía esta señal a un demonio, se reiniciará y releerá su archivo de configuración. El servidor web Apache es un ejemplo de demonio que usa la señal `HUP` de esta forma.
2	`INT`	Interrupción. Realiza la misma función que las teclas `Ctrl-c` enviadas desde el terminal. Normalmente termina un programa.

9	`KILL`	Mata. Esta señal es especial. Mientras que los programas pueden elegir manejar las señales que les son enviadas de diferentes formas, incluso ignorarlas todas, la señal `KILL` realmente nunca se envía al programa objetivo. En su lugar, el kernel inmediatamente termina el proceso. Cuando un proceso es finalizado de esta forma, no tiene oportunidad de "limpiarse" por sí mismo o salvar su trabajo. Por esta razón, la señal `KILL` sólo debería ser utilizada como último recurso cuando otras señales de finalización fallan.
15	`TERM`	Terminar. Esta es la señal por defecto enviada por el comando kill. Si un programa aún está lo suficientemente "vivo" como para recibir señales, se terminará.
18	`CONT`	Continuar. Esta restaurará un proceso después de una señal `STOP`.
19	`STOP`	Parar. Esta seña causa la pausa de un proceso sin terminarlo. Como la señal `KILL`, no se envía al proceso objetivo, y por lo tanto no puede ser ignorada.

Probemos el comando `kill`:

```
[me@linuxbox ~]$ xlogo &[1] 13546[me@linuxbox ~]$ kill -1
13546[1]+ Hangup xlogo
```

En este ejemplo, arrancamos el programa `xlogo` en el fondo y luego le enviamos una señal `HUP` con `kill`. El programa `xlogo` termina y el shell indica que el proceso en el fondo ha recibido una señal de cuelgue. Necesitarás pulsar la tecla intro un par de veces antes de ver el mensaje. Fíjate que las señales pueden ser especificadas con número o con nombre, incluyendo el nombre precedido de las letras "SIG":

```
[me@linuxbox ~]$ xlogo &[1] 13601[me@linuxbox ~]$ kill -INT
13601[1]+ Interrupt xlogo[me@linuxbox ~]$ xlogo &[1]
13608[me@linuxbox ~]$ kill -SIGINT 13608[1]+ Interrupt
xlogo
```

Repite el ejemplo anterior y prueba las otras señales. Recuerda, también puedes usar jobspecs en lugar de PIDs.

Los procesos, como los archivos, tienen propietarios, y debes ser el propietario de un proceso (o el superusuario) para enviarle señales con `kill`.

Además de la lista de señales anterior, cuya mayoría se usa con `kill`, hay otras señales usadas frecuentemente por el sistema. Aquí tenemos una lista de otras señales comunes:

Tabla 10-5: Otras señales comunes

Número	Nombre	Significado
3	`QUIT`	Quit.
11	`SEGV`	Violación de segmentación. Esta señal se envía si un programa hace un uso ilegal de la memoria, o sea, trata de escribir en un sitio en el que no está autorizado.

20	TSTP	Stop de terminal. Esta es la señal enviada por el terminal cuando se pulsa Ctrl-z. Al contrario que la señal STOP, la señal TSTP la recibe el programa pero el programa puede elegir ignorarla.
28	WINCH	Cambio de ventana. Es una señal enviada por el sistema cuando una ventana cambia de tamaño. Algunos programas, como top y less responden a esta señal redibujándose a sí mismos para ajustarse a las nuevas dimensiones de la ventana.

Para los curiosos, una lista completa de las señales se puede ver con el siguiente comando:

```
[me@linuxbox ~]$ kill -l
```

Enviando señales a múltiples procesos con killall

También es posible enviar señales a múltiples procesos eligiendo un programa específico o un nombre de usuario usando el comando killall. Aquí tenemos la sintaxis:

```
killall [-u user] [-signal] name...
```

Para demostrarlo, iniciaremos un par de instancias del programa xlogo y las finalizaremos:

```
[me@linuxbox ~]$ xlogo &[1] 18801[me@linuxbox ~]$ xlogo
&[2] 18802[me@linuxbox ~]$ killall xlogo[1]- Terminated
xlogo[2]+ Terminated xlogo
```

Recuerda, como con kill, debes tener privilegios de superusuario para enviar señales a procesos que no te pertenecen.

Más comandos relacionados con procesos

Como monitorizar procesos es una tarea importante para la administración de sistemas, hay muchos comandos para ello. Aquí tenemos algunos para jugar con ellos:

Tabla 10-6: Otros comandos relacionados con procesos

Comando	Descripción
pstree	Muestra una lista de procesos dispuestos en una estructura de árbol mostrando las relaciones padre/hijo entre los procesos.
vmstat	Muestra una instantánea de los recursos de sistema usados, incluyendo memoria, intercambio e I/O de disco. Para ver una pantalla continua, añade al comando un periodo de tiempo (en segundos) para las actualizaciones. Por ejemplo: vmstat 5. Finaliza la salida con Ctrl-c.
xload	Un programa gráfico que traza un gráfico mostrando la carga del sistema a lo largo del tiempo.
tload	Igual que el programa xload, pero dibuja el gráfico en el terminal. Finaliza la salida con Ctrl-c.

Resumiendo

Los sistemas más modernos proporcionan un mecanismo para manejar múltiples procesos. Linux provee un rico catálogo de herramientas para este propósito. Dado que Linux es el sistema operativo de servidores más expandido en el mundo, ésto tiene mucho sentido. Sin embargo, al contrario que otros sistemas, Linux se basa principalmente en herramientas de línea de comandos para el manejo de procesos. Aunque hay herramientas gráficas para procesos en Linux, las herramientas de línea comandos son ampliamente preferidas por su velocidad y ligereza. Aunque las herramientas de la GUI son más bonitas, a menudo crean una gran carga en el sistemas ellas mismas, lo cual estropean el propósito.

El entorno

Como vimos antes, el shell mantiene una cantidad de información durante nuestra sesión de shell llamada el *entorno*. Los datos almacenados en el entorno los usan los programas para determinar cosas acerca de nuestra configuración. Mientras que la mayoría de programas usan *archivos de configuración* para almacenar las configuraciones del programa, algunos programas también buscan valores almacenados en el entorno para ajustar su comportamiento. Sabiendo ésto, podemos usar el entorno para personalizar nuestra experiencia con el shell.

En éste capítulo, trabajaremos con los siguientes comandos:

- `printenv` – Imprime parte o todo el entorno
- `set` – Establece opciones del shell
- `export` – Exporta el entorno a los programas que se ejecuten a continuación
- `alias` – Crea un alias para un comando

¿Qué se almacena en el entorno?

El shell almacena dos tipos básicos de datos en el entorno, con `bash`, los dos tipos son casi indistinguibles. Hay *variables de entorno* y *variables de shell*. Las variables de shell son bits de datos puestos allí por `bash`, y las variables de entorno son básicamente todo lo demás. Además de las variables, el shell también almacena algunos datos programáticos, llamados *alias* y *funciones de shell*. Veremos los alias en el Capítulo 5, y las funciones de shell (que están relacionadas con los scripts de shell) las veremos en la Parte 4.

Examinando el entorno

Para ver que hay almacenado en el entorno, podemos usar el set integrado en `bash` o el programa `printenv`. El comando set nos mostrará tanto las variables del shell como las del entorno, mientras que `printenv` sólo mostrará éstas últimas. Como nuestra lista de contenidos del entorno será muy larga, es mejor pasar la salida de cada comando por un pipe hacia `less`:

```
[me@linuxbox ~]$ printenv | less
```

Haciéndolo, deberíamos obtener algo parecido a ésto:

```
KDE_MULTIHEAD=falseSSH_AGENT_PID=6666HOSTNAME=linuxboxGPG_A
GENT_INFO=/tmp/gpg-PdOt7g/S.gpg-
agent:6689:1SHELL=/bin/bashTERM=xtermXDG_MENU_PREFIX=kde-
HISTSIZE=1000XDG_SESSION_COOKIE=6d7b05c65846c3eaf3101b0046b
d2b00-1208521990.996705-1177056199GTK2_RC_FILES=/etc/gtk-
2.0/gtkrc:/home/me/.gtkrc-
2.0:/home/me/.kde/share/config/gtkrc-
2.0GTK_RC_FILES=/etc/gtk/gtkrc:/home/me/.gtkrc:/home/me/.kd
```

```
e/share/config/gtkrcGS_LIB=/home/me/.fontsWINDOWID=29360136
QTDIR=/usr/lib/qt-3.3QTINC=/usr/lib/qt-
3.3/includeKDE_FULL_SESSION=trueUSER=meLS_COLORS=no=00:fi=0
0:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01:cd=40;33;
01:or=01;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:*.exe
:
```

Lo que vemos es una lista de las variables de entorno y sus valores. Por ejemplo, vemos una variable llamada `USER`, que contiene el valor "me". El comando `printenv` puede listar también el valor de una variable específica:

```
[me@linuxbox ~]$ printenv USERme
```

El comando set, cuando lo usamos sin opciones o argumentos, mostrará tanto las variables del shell como las del entorno, así como cualquier función definida del shell. Al contrario de `printenv`, su salida está cortésmente ordenada alfabéticamente:

```
[me@linuxbox ~]$ set | less
```

También es posible ver el contenido de una variable usando el comando `echo`, así:

```
[me@linuxbox ~]$ echo $HOME/home/me
```

Un elemento del entorno que ni `set` ni `printenv` muestran son los `alias`. Para verlos, introduce el comando alias sin argumentos:

```
[me@linuxbox ~]$ aliasalias l.='ls -d .* --color=tty'alias
ll='ls -l --color=tty'alias ls='ls --color=tty'alias
vi='vim'alias which='alias | /usr/bin/which --tty-only
--read-alias --showdot--show-tilde'
```

Algunas variables interesantes

El entorno contiene unas pocas variables, y sin embargo tu entorno puede diferir del presentado aquí, verás más o menos las siguientes variables en tu entorno:

Tabla 11-1: Variables de entorno

Variable	Contenidos
`DISPLAY`	El nombre de tu pantalla si estás corriendo un entorno gráfico. Normalmente es ":0", que significa la primera pantalla generada por el servidor X.
`EDITOR`	El nombre del programa que se usa para editar texto.
`SHELL`	El nombre de tu programa de shell
`HOME`	La ruta de tu directorio home.
`LANG`	Define la configuración de caracteres y el orden de colación de tu idioma.
`OLD_PWD`	El anterior directorio de trabajo.
`PAGER`	El nombre del programa que se usa para paginar la salida. A menudo está establecido en `/usr/bin/less`.
`PATH`	Una lista separada por puntos de los directorios en que se busca cuando introduces el nombre de un programa ejecutable.

PS1	Prompt String 1. Define el contenido de tu prompt de shell. Como veremos más tarde, puede ser ampliamente personalizado.
PWD	El directorio de trabajo actual.
TERM	El nombre de tu tipo de terminal. Los sistemas como-Unix soportan muchos protocolos de terminal; esta variable establece el protocolo que se va a usar con tu emulador de terminal.
TZ	Especifica tu zona horaria. La mayoría de los sistemas como-Unix mantienen un reloj interno en el ordenador en *Coordinated Universal Time* (UTC) y luego muestran la hora local aplicando un desfase especificado por esta variable.
USER	Tu nombre de usuario.

No te preocupes si algunas de estas variables faltan. Varían según la distribución.

¿Cómo se establece el entorno?

Cuando nos identificamos en el sistema, el programa `bash` arranca, y lee una serie de scripts de configuración llamados *archivos de arranque (startup files)*, que definen el entorno por defecto compartido para todos los usuarios. A continuación se leen más archivos de arranque en tu directorio home que definen tu entorno personal. La secuencia exacta depende del tipo de sesión de shell que se inicie. Hay dos tipos: sesión de shell con login y sesión de shell sin login.

Una sesión de shell con login es aquella en la que nos preguntan nuestro nombre de usuario y contraseña; cuando arrancamos una sesión de consola virtual, por ejemplo. Un sesión de shell sin login es la que típicamente se da cuando arrancamos una sesión de terminal en la GUI.

Los login de shell leen uno o más archivos de arranque como se muestra en la Tabla 11-2:

Tabla 11-2: Archivos de arranque para sesiones de shell con login

Archivo	Contenidos
/etc/profile	Un script configuración global que se aplica a todos los usuarios.
~/.bash_profile	Un archivo de arranca personal del usuario. Puede utilizarse para extender o sobrescribir configuraciones del script de configuración global.
~/.bash_login	Si `~/.bash_profile` no se encuentra, `bash` intenta leer este script.
~/.profile	Si no se encuentran ni `~/.bash_profile` ni `~/.bash_login`, `bash` intenta leer este archivo. Éste es el que vieme por defecto en las distribuciones basadas en Debian, como Ubuntu.

Las sesiones de shell sin login leen los siguientes archivos de arranque:

Tabla 11-3: Archivos de arranque para sesiones de shell sin login

Archivo	Contenidos
`/etc/bash.bashrc`	Un script de configuración global que se aplica a todos los usuarios
`~/.bashrc`	Un archivo de arranque personal del usuario. Puede usarse para extender o sustituir las configuraciones del script de configuración global.

Además de leer los anteriores archivos de arranque, los shells sin login también heredan el entorno de sus procesos padre, normalmente un shell con login.

Echa un vistazo a tu sistema y mira cuales de estos archivos de arranque tienes. Recuerda – como la mayoría de los archivos listados anteriormente empiezan con un punto (que significa que están ocultos), necesitarás usar la opción "-a" cuando uses `ls`.

El archivo `~/.bashrc` es probablemente el archivo de arranque más importante desde el punto de vista de un usuario normal, ya que casi siempre se lee. Los shell sin login lo leen por defecto y la mayoría de archivos de arranque para los shell con login se escriben de forma que lean el archivo `~/.bash_login` también.

¿Qué hay en un archivo de arranque?

Si miramos dentro de una archivo `.bash_profile` típico (tomado de un sistema CentOS), vemos algo así:

```
# .bash_profile# Get the aliases and functionsif [ -f
~/.bashrc ]; then. ~/.bashrcfi# User specific environment
and startup programsPATH=$PATH:$HOME/binexport PATH
```

Las líneas que empiezan con un "#" son comentarios y no son leidas por el shell. Son para que sean legibles por personas. La primera cosa interesante ocurre en la cuarta línea, con el siguiente código:

```
if [ -f ~/.bashrc ]; then. ~/.bashrcfi
```

Ésto se llama un *comando if compuesto*, que veremos más en profundidad cuando lleguemos al scripting de shell en la Parte 4, pero por ahora lo traduciremos:

```
If the file "~/.bashrc" exists, thenread the "~/.bashrc"
file.
```

Podemos ver que este fragmento de código es la forma en que un shell con login toma los contenidos de `.bashrc`. Lo siguiente en nuestros archivo de arranque tiene que ver con la variable `PATH`.

¿Te has preguntado alguna vez como sabe el shell donde encontrar los comandos cuando los introducimos en la linea de comandos? Por ejemplo, cuando introducimos ls, el shell no busca en todo el ordenador para encontrar `/bin/ls` (la ruta completa del comando ls), en su lugar, busca en una lista de directorios que están contenidos en la variable `PATH`.

La variable `PATH` a menudo (pero no siempre, depende de la distribución) es establecida por el archivo de arranque `/etc/profile` con éste código:

```
PATH=$PATH:$HOME/bin
```

PATH se modifica para añadir el directorio $HOME/bin al final de la lista. Ésto es un ejemplo de expansión con parámetros, como vimos en el Capítulo 7. Para demostrar como funciona, prueba lo siguiente:

```
[me@linuxbox ~]$ foo="This is some "[me@linuxbox ~]$ echo
$fooThis is some[me@linuxbox ~]$
foo=$foo"text."[me@linuxbox ~]$ echo $fooThis is some text.
```

Usando esta técnica, podemos añadir texto al final de los contenidos de una variable. Añadiendo la cadena $HOME/bin al final del contenido de la variable PATH, el directorio $HOME/bin se añade a la lista de directorios a buscar cuando se introduce un comando. Ésto significa que cuando queremos crear un directorio dentro de nuestro directorio home para almacenar nuestros programas privados, el shell está preparado para contenerlos. Todo lo que tenemos que hacer es llamarlo bin, y estamos listos para empezar.

Nota: Muchas distribuciones proveen esta configuración de PATH por defecto. Algunas distribuciones basadas en Debian, como Ubuntu, buscan la existencia del directorio ~/bin en el login, y dinámicamente lo añaden a la variable PATH si encuentran el directorio.

Finalmente, tenemos:

```
export PATH
```

El comando export le dice al shell que haga que los contenidos de PATH estén disponibles para los pro esos hijos de este shell.

Modificando el entorno

Como ya sabemos donde están los archivos de arranque y que contienen, podemos modificarlos para personalizar nuestro entorno.

¿Qúe archivos podríamos modificar?

Como regla general, para añadir directorios a tu PATH, o definir variables de entorno adicionales, coloca dichos cambios en .bash_profile (o equivalente, según tu distribución. Por ejemplo, Ubuntu usa .profile.) Para todo lo demás, coloca los cambios en .bashrc. A menos que seas el administrador del sistema y necesites cambiar los predefinidos para todos los usuarios del sistema, reduce tus modificaciones a los archivos de tu directorio home. En realidad es posible cambiar los archivos en /etc como profile, y en muchos casos sería sensato hacerlo, pero por ahora, juguemos con seguridad.

Editores de texto

Para editar (p.ej., modificar) los archivos de arranque de shell, así como la mayoría del resto de configuraciones del sistema, usamos un programa llamado *editor de texto*. Un editor de texto es un programa que es, de algún modo, como un procesador de texto en que podemos editar las palabras que hay en la pantalla con un movimiento del cursor. Difiere de un procesador de texto en que sólo soporta texto plano, y a menudo contiene funciones diseñadas para programas de escritura. Los editores de texto son la herramienta central usada por los desarrolladores de software para escribir código, y por los administradores de sistema para manejar los archivos de configuración que controlan el sistema.

Hay muchos editores de textos diferentes disponibles para Linux; tu sistema probablemente tenga varios instalados. ¿Por qué hay tantos diferentes? Probablemente porque a los programadores les gusta escribirlos, y como los programadores los usan intensamente, escriben editores para expresar sus propios deseos sobre cómo deberían funcionar.

Los editores de texto se clasifican en dos categorías básicas: gráficos y basados en texto. GNOME y KDE incluyen algunos editores gráficos populares. GNOME cuenta con un editor llamado `gedit`, que usualmente es llamado "Editor de Texto" en el menú de GNOME. KDE normalmente cuenta con tres que son (en orden de complejidad creciente) `kedit`, `kwrite` y `kate`.

Hay muchos editores basados en texto. Los más populares que encontrarás son `nano`, `vi`, y `emacs`. El editor `nano` es un editor simple y fácil de usar diseñado como un sustituto del editor `pico` proporcionado por la suite de email PINE. El editor `vi` (en la mayoría de los sistemas Linux sustituido por un programa llamado `vim`, que es la abreviatura de "Vi Improved" - "Vi mejorado") es el editor tradicional de los sistemas como-Unix. Será el objeto de nuestro próximo capítulo. El editor emacs fue originalmente escrito por Richard Stallman. Es un entorno de programación gigantesco, multi-propósito y hacelo-todo. Aunque está fácilmente disponible, rara vez viene instalado en los sistemas Linux por defecto.

Usando un editor de texto

Todos los editores de texto pueden ser llamados desde la línea de comandos escribiendo el nombre del editor seguido del nombre del archivo que queremos editar. Si el archivo todavía no existe, el editor asume que queremos crear un nuevo archivo. Aquí tenemos un ejemplo usando `gedit`:

```
[me@linuxbox ~]$ gedit some_file
```

Este comando arrancará el editor de texto `gedit` y cargará el archivo llamado "some_file", si existe.

Todos los editores gráficos son muy auto-explicativos, así que no los veremos aquí. En su lugar, nos concentraremos en nuestro primer editor de texto basado en texto, `nano`. Arranquemos `nano` y editemos el archivo `.bashrc`. Pero antes de hacerlo, practiquemos algo de "informática segura". Cada vez que editamos un archivo de configuración importante, siempre es una buena idea crear una copia de seguridad del archivo primero. Esto nos protege en caso de que estropeemos el archivo mientras lo editamos. Para crear una copia de seguridad del archivo `.bashrc`, haz esto:

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

No importa como llames al archivo de copia de seguridad, sólo escoge un nombre comprensible. Las extensiones ".bak", ".sav", ".old" y ".orig" son las formas más populares de indicar que es un archivo de copia de seguridad. Ah, y recuerda que `cp` sobrescribirá los *archivos existentes silenciosamente*.

Ahora que tenemos un archivo de copia de seguridad, arrancaremos el editor:

```
[me@linuxbox ~]$ nano .bashrc
```

Una vez que nano arranca, tendremos una pantalla como esta:

```
GNU nano 2.0.3 File: .bashrc
# .bashrc
```

```
# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi
# User specific aliases and functions









[ Read 8 lines ]
^G Get Help^O WriteOut^R Read Fil^Y Prev Pag^K Cut Text^C Cur Pos
^X Exit ^J Justify ^W Where Is^V Next Pag^U UnCut Te^T To Spell
```

Nota: si tu sistema no tiene `nano` instalado, puedes usar el editor gráfico en su lugar.

La pantalla consiste en un encabezado arriba, el texto del archivo que estamos editando en el centro y un menú de comandos abajo. Como `nano` fue diseñado para reemplazar el editor de texto disponible en en cliente de correo, es muy escaso en opciones de edición.

El primer comando que deberías aprender en cualquier editor de texto es cómo salir del programa. En el caso de `nano`, pulsas `Ctrl-x` para salir. Esto viene indicado en el menú de abajo de la pantalla. La notación “^X” significa `Ctrl-x`. Ésta es una notación común para los caracteres de control usados por muchos programas.

El segundo comando que necesitamos conocer es cómo salvar nuestro trabajo. Con nano es `Ctrl-o`. Con este conocimiento en nuestro haber estamos preparados para hacer algo de edición. Usando la tecla de la flecha hacia abajo y/o el tecla `PageDown`, mueve el cursor al final del archivo, y luego añade las siguientes lineas al archivo `.bashrc`:

```
umask 0002export HISTCONTROL=ignoredupsexport
HISTSIZE=1000alias l.='ls -d .* --color=auto'alias ll='ls
-l –color=auto'
```

Nota: Tu distribución podría incluir ya alguna de ellas, pero el duplicarlas no hace daño a nadie. Aquí tenemos el significado de lo que hemos añadido:

Tabla 11-4: Lineas añadidas a nuestro .bashrc

Línea	Significado
`umask 0002`	Establece el umask para resolver el problema con los directorios compartidos que vimos en el Capítulo 9.
`export HISTCONTROL=ignoredups`	Hace que la función de grabación del historial del shell ignore un comando si el mismo comando ya ha sido grabado.
`export HISTSIZE=1000`	Aumenta el tamaño del historial de comandos del predefinido de 500 lineas a 1000 lineas.

`alias l.='ls -d .* --color=auto'`	Crea un nuevo comando llamado "`l.`" que muestra todos las entradas de directorio que empiezan con un punto.
`alias ll='ls -l --color=auto'`	Crea un nuevo comando llamado "`ll`" que muestra un listado de directorios en formato largo.

Como podemos ver, muchas de nuestras lineas añadidas no son intuitivas ni obvias, así que sería una buena idea añadir comentarios a nuestro archivo `.bashrc` para ayudarnos a explicar las cosas a los humanos.

Usando el editor, cambiamos nuestras lineas añadidas para que quede así:

```
# Change umask to make directory sharing easierumask 0002# Ignore duplicates in command history and increase# history size to 1000 linesexport HISTCONTROL=ignoredupsexport HISTSIZE=1000# Add some helpful aliasesalias l.='ls -d .* --color=auto'alias ll='ls -l –color=auto'
```

¡Ah, mucho mejor! Con nuestros cambios completados, pulsa `Ctrl-o` para guardar nuestro archivo .bashrc modificado, y `Ctrl-x` para salir de `nano`.

¿Por qué son importantes los comentarios?

Siempre que modificas archivos de configuración es una buena idea añadir algunos comentarios para documentar tus cambios. Seguro que recordarás lo que has cambiado mañana, pero ¿qué pasará dentro de seis meses? Hazte un favor y añade algunos comentarios. Mientras estás en ello, no es mala idea guardar un registro de los cambios que haces.

Los scripts de shell y los archivos de arranque de `bash` usan el símbolo "#" para comenzar un comentario. Otros archivos de configuración pueden usar otros símbolos. La mayoría de los archivos de configuración tendrán comentarios. Úsalos como guía.

A menudo verás líneas en archivos de configuración que está *comentadas* para evitar que sean usadas por el programa afectado. Ésto se hace para dar al lector recomendaciones para posibles configuraciones o ejemplos de la sintaxis de configuración correcta. Por ejemplo, el archivo `.bashrc` de Ubuntu 8.04 contiene estas líneas:

```
# some more ls aliases#alias ll='ls -l'#alias la='ls -A'#alias l='ls -CF'
```

Las últimas tres lineas son definiciones válidas de alias que han sido comentadas. Si eliminas los símbolos "#" de delante de las tres lineas, una técnica llamada *descomentar*, activarás los alias. Inversamente, si añades un símbolo "#" al principio de una linea, puedes desactivar una linea de configuración mientras que conservas la información que contiene.

Activando nuestros cambios

Los cambios que hemos hecho a nuestro `.bashrc` no tendrán efecto hasta que cerremos nuestra sesión de terminal y arranquemos una nueva, ya que el archivo `.bashrc` sólo se lee al principio de la sesión. Sin embargo, podemos forzar que bash relea el `.bashrc` modificado con el siguiente

comando:

```
[me@linuxbox ~]$ source .bashrc
```

Después de hacer esto, deberíamos poder ver el efecto de nuestros cambios. Prueba uno de los nuevos alias:

```
[me@linuxbox ~]$ ll
```

Resumiendo

En este capítulo hemos aprendido una habilidad esencial – editar archivos de configuración con un editor de texto. Continuando, tal como leemos man pages para los comandos, toma nota de las variables de entorno que soportan los comandos. Puede haber un par de joyas. En capítulos posteriores, aprenderemos las funciones de shell, una característica poderosa que podrás incluir en los archivos de arranque de `bash` y añadir a tu arsenal de comandos personalizados.

Para saber más

- La sección INVOCATION de la man page de `bash` cubre los archivos de arranque de bash al detalle.

Una cómoda introducción a vi

Hay un viejo chiste sobre alguien que visita New York City y pregunta a un transeúnte cómo ir al sitio más famoso de música clásica:

Visitante: Perdone, ¿cómo puedo llegar al Carnegie Hall?
Transeúnte: ¡Practique, practique, practique!

Aprender la línea de comandos de Linux, como llegar a ser un pianista experto, no es algo que logramos en una tarde. Lleva años de práctica. En este capítulo, presentaremos el editor de texto `vi` (que se pronuncia "vi ai"), uno de los programas clave en la tradición Unix. `vi` es famoso por su compleja interfaz de usuario, pero cuando vemos a un maestro sentado en el teclado y comienza a "tocar", seremos testigos de algo grandioso. No nos convertiremos en maestros en este capítulo, pero cuando lo terminemos, aprenderemos como usar los "palillos" en `vi`.

Por qué deberíamos aprender vi

En esta moderna era de los editores gráficos y los editores de texto faciles-de-usar como nano, ¿por qué deberíamos aprender `vi`? Hay tres buenas razones:

- `vi` está siempre disponible. Puede ser un salvavidas si tenemos un sistema sin interfaz gráfica, como un servidor remoto o un sistema local con la configuración de las X estropeada. `nano`, aunque cada vez es más popula,r todavía no es universal. POSIX, un estándar de compatibilidad para sistemas Unix, requiere que vi está presente.
- `vi` es ligero y rápido. Para muchas tareas, es más fácil arrancar `vi` que encontrar el editor gráfico de texto en los menús y esperar que sus múltiples megabytes carguen. Además, `vi` está diseñado para teclear rápido. Como veremos, un usuario o usuaria con experiencia en `vi` nunca tiene que separar sus dedos del teclado mientras está editando.
- No queremos que otros usuarios de Linux y Unix piensen que somos unos gallinas.

Ok, a lo mejor sólo son dos buenas razones.

Un poco de historia

La primera versión de `vi` fue escrita en 1976 por Bill Joy, un estudiante de la Universidad de California en Berkley que más tarde se convirtió en co-fundador de Sun Microsystems. vi toma su nombre de la palabra "visual", porque estaba pensado para permitir la edición en un terminal de vídeo con un cursor con movimiento. Anteriormente a los *editores visuales*, había *editores de línea* para ir a una línea concreta y decirle los cambios a realizar, como añadir o eliminar texto. Con la llegada de los terminales de vídeo (en lugar de los terminales basados en impresión como los teletipos) los editores visuales se hicieron posibles. `vi` en realidad incorpora un potente editor de líneas llamado `ex`, y podemos usar la edición de líneas de comandos mientras usamos `vi`.

La mayoría de las distribuciones no incluyen el auténtico `vi`; en su lugar, están equipados con un sustituto mejorado llamado `vim` (que es la abreviatura de "vi improved " - "vi mejorado") escrito por Bram Moolenaar. `vim` es una mejora sustancial sobre el `vi` tradicional de Unix y normalmente está simbólicamente enlazado (o con un alias) al nombre "vi" en los sistemas Linux. En los temas que siguen asumiremos que tenemos un programa llamado "vi" que en realidad es `vim`.

Arrancando y parando vi

Para arrancar `vi`, simplemente escribe lo sisguiente:

```
[me@linuxbox ~]$ vi
```

Y aparecerá una pantalla como ésta:

```
~~~ VIM - Vi Improved~~ version 7.1.138~ by Bram Moolenaar
et al.~ Vim is open source and freely distributable~~
Sponsor Vim development!~ type :help sponsor<Enter> for
information~~ type :q<Enter> to exit~ type :help<Enter> or
<F1> for on-line help~ type :help version7<Enter> for
version info~~ Running in Vi compatible mode~ type :set
nocp<Enter> for Vim defaults~ type :help cp-default<Enter>
for info on this~~~
```

Tal como hicimos con `nano` antes, la primera cosa que aprenderemos es cómo salir. Para salir, introducimos el siguiente comando (fíjate que los dos puntos forman parte del comando):

```
:q
```

El prompt del shell debería volver. Si, por alguna razón, `vi` no se cierra (normalmente porque hacemos un cambio a un archivo que no ha sido salvado todavía), podemos decirle a `vi` que eso es lo que queremos añadiendo un signo de exclamación al comando:

```
:q!
```

Consejo: Si te "pierdes" en `vi`, prueba a pulsar la tecla `Esc` dos veces para encontrar el camino otra vez.

Modo de compatibilidad

En la pantalla de inicio del ejemplo anterior (tomada de Ubuntu 8,04), vemos el texto "Running in Vi compatible mode.". Ésto significa que `vim` funcionará en un modo que es parecido al funcionamiento normal de `vi` en lugar del funcionamiento mejorado de `vim`. Para los objetivos de este capítulo, querremos ejecutar vim con su funcionamiento mejorado.

Para hacerlo, tenemos unas pocas opciones:

Prueba ejecutar `vim` en lugar de `vi`.

Si funciona, considera añadir `alias vi='vim'` a tu archivo `.bashrc`.

Alternativamente, usa este comando para añadir una linea al archivo de configuración de vim:

```
echo “set nocp” >> ~/.vimrc
```

Las diferentes distribuciones de Linux empaquetan `vim` de formas distintas. Algunas distribuciones instalan una versión mínima de `vim` par defecto que sólo soporta un catálogo limitado de funciones de `vim`. Según vayamos viendo las siguientes lecciones, encontrarás que faltan funciones. Si ese es el caso, instala la versión completa de `vim`.

Modo de edición

Arranquemos `vi` de nuevo, esta vez pasándole el nombre de un archivo inexistente. Así es como creamos un nuevo archivo con `vi`:

```
[me@linuxbox ~]$ rm -f foo.txt[me@linuxbox ~]$ vi foo.txt
```

Si todo va bien, deberíamos tener una pantalla como esta:

```
~~~~~~~~~~~~~~~~~~~~~"foo.txt" [New File]
```

Las tildes de ñ ("~") indican que no existe texto en esa línea. Ésto muestra que tenemos un archivo vacío. **¡No escribas nada todavía!**

La segunda cosa más importante a aprender sobre `vi` (después de aprender a salir) es que `vi` es un *editor modal*. Cuando `vi` arranca, lo hace en *modo comando.* En este modo, casi cualquier tecla es un comando, así que si empezáramos a escribir, `vi` básicamente se volvería loco y se formaría un gran lio.

Entrar en el modo de inserción

Para añadir texto a nuestro archivo, debemos entrar primero en el *modo de inserción*. Para hacerlo, pulsamos la tecla "i". Tras ésto, deberíamos ver lo siguiente en la parte inferior de la pantalla si `vim` está funcionando en su modo normal mejorado (esto no aparecerá en el modo compatible con `vi`):

```
-- INSERT –
```

Ahora podemos introducir texto. Prueba ésto:

```
The quick brown fox jumped over the lazy dog.
```

Para salir del modo de inserción y volver al modo de comandos, pulsa la tecla `Esc`:

Salvando nuestro trabajo

Para salvar el cambio que acabamos de hacer en nuestro archivo, sólo tenemos que introducir un *comando ex* estando en el modo comando. Ésto se hace fácilmente pulsando la tecla ":". Después de hacer ésto, un carácter de dos puntos aparecerá en la parte de abajo de la pantalla:

```
:
```

Para guardar los cambios en nuestro archivo modificado, añadiremos una "w" después de los dos puntos y pulsaremos Enter:

```
:w
```

El archivo será guardado en el disco duro y tendremos un mensaje de confirmación en la parte inferior de la pantalla, como éste:

```
"foo.txt" [New] 1L, 46C written
```

Consejo: Si lees la documentación de `vim`, notarás que (confusamente) el modo comando se llama *modo normal* y los comandos ex se llaman *modo comando*. Ten cuidado.

Moviendo el cursor

Mientras que está en el modo comando, `vi` ofrece un gran número de comandos de movimiento, algunos de los cuales comparte con `less`. Aquí hay una muestra:

Tabla 12-1: Teclas de movimiento de cursor

Tecla	Mueve el cursor
`l` o Flecha derecha	Un carácter a la derecha.
`h` o flecha izquierda	Un carácter a la izquierda.
`j` o Flecha abajo	Abajo una línea.
`k` o Flecha arriba	Arriba una línea.
`0` (cero)	Al principio de la linea actual.
`^`	Al primer espacio no en blanco en la linea actual.
`$`	Al final de la línea actual.
`w`	Al principio de la siguiente palabra o signo de puntuación
`W`	Al principio de la siguiente palabra, ignorando signos de puntuación.
`b`	Al principio de la anterior palabra o signo de puntuación.
`B`	Al principio de la anterior palabra, ignorando signos de puntuación.
`Ctrl-f` o `Page Down`	Abajo una página.
`Ctrl-b` o `Page Up`	Arriba una página.
`number``G`	A la línea *número*. Por ejemplo, 1G lo mueve a la primera linea del archivo.

`G`	A la última línea del archivo.

¿Por qué se usan las teclas `h`, `j`, `k` y `l` para movimientos de cursor? Porque cuando `vi` fue originalmente escrito, no todos los terminales de vídeo tenían las teclas de las flechas, y los hábiles mecanógrafos podían usar teclas normales del teclado para mover el cursor sin tan siguiera tener que despegar los dedos del teclado.

Muchos comandos en vi pueden ir precedidos con un número, tal como el comando "`G`" listado antes. Precediendo un comando con un número, podemos especificar el número de veces que un comando va a ser ejecutado. Por ejemplo, el comando "`5j`" hace que `vi` mueva el cursor hacia abajo cinco líneas.

Edición básica

La mayoría de las ediciones consisten en unas pocas operaciones básicas como insertar texto, borrar texto y mover texto de sitio cortando y pegando. `vi`, por supuesto, soporta todas estas operaciones en su forma única. `vi` también proporciona una forma limitada de deshacer. Si pulsamos la tecla "u" mientras está en modo comando, `vi` deshará el último cambio que hayas hecho. Ésto será práctico cuando probemos algunos de los comandos de edición básica.

Añadiendo texto

`vi` tiene varias formas para entrar en el modo inserción. Ya hemos usado el comando `i` para insertar texto.

Volvamos a nuestro archivo `foo.txt` un momento:

```
The quick brown fox jumped over the lazy dog.
```

Si quisiéramos añadir texto al final de esta frase, descubriríamos que el comando `i` no lo hace, ya que no podemos mover el cursor más allá del final de la línea. `vi` cuenta con un comando para añadir texto, el comando sensatamente llamado "a". Si movemos el cursor al final de la línea y escribimos "a", el cursor se moverá hasta pasado el final de la línea y `vi` entrará en modo inserción. Ésto nos permitirá añadir más texto:

```
The quick brown fox jumped over the lazy dog. It was cool.
```

Recuerda pulsar la tecla `Esc` para salir del modo inserción.

Como casi siempre querremos añadir texto al final de una línea, `vi` ofrece un atajo para moverte al final de la línea actual y comenzar a añadir texto. Es el comando "A". Probémoslo y añadamos algunas líneas más a nuestro archivo.

Primero, movamos el cursor al principio de la línea usando el comando "0" (cero).

Ahora pulsamos "A" y añadimos las siguientes líneas de texto:

```
The quick brown fox jumped over the lazy dog. It was cool.

Line 2

Line 3
```

```
Line 4

Line 5
```

De nuevo, pulsamos la tecla `Esc` para salir del modo de inserción.

Como podemos ver, el comando “A” es más útil que mover el cursor al final de la línea antes de entrar en el modo de inserción.

Abriendo una línea

Otra forma en que podemos insertar texto es “abriendo” una línea. Ésto inserta una línea en blanco entre dos líneas existentes y entra en el modo inserción. Tiene dos variantes:

Tabla 12-2: Teclas de apertura de líneas

Comando	Abre
`o`	La línea bajo la línea actual.
`O`	La línea sobre la línea actual.

Podemos demostrar esto de la siguiente forma: coloca el cursor en la “Línea 3” y luego pulsa la tecla o.

```
The quick brown fox jumped over the lazy dog. It was cool.

Line 2

Line 3



Line 4

Line 5
```

Una nueva línea se abre bajo la tercera línea y entramos en el modo inserción. Salimos del modo inserción pulsando la tecla `Esc`. Pulsamos la tecla u para deshacer nuestro cambio.
Pulsa la tecla `O` para abrir la línea sobre el cursor.

```
The quick brown fox jumped over the lazy dog. It was cool.

Line 2

Line 3

Line 4

Line 5
```

Salimos del modo inserción presionando la tecla `Esc` y deshaciendo nuestro cambio pulsando `u`.

Borrando texto

Como esperaríamos, `vi` ofrece una variedad de formas de borrar texto, todos ellos contienen uno o dos atajos de teclado. Primero, la tecla `x` borrará un carácter en la localización del cursor. `x` debería ir precedida por un número especificando cuantos caracteres van a ser borrados. La tecla `d` es de propósito más general. Como `x`, debería ir precedida por un número especificando el número de veces que se realizará el borrado. Además, `d` siempre va seguido por un comando de movimiento que controla el tamaño del borrado. Aquí tenemos algunos ejemplos:

Tabla 12-3: Comandos de borrado de texto

Comando	Borra
`x`	El carácter actual.
`3x`	El carácter actual y los siguientes dos caracteres.
`dd`	La línea actual.
`5dd`	La línea actual y las cuatro siguientes.
`dW`	Desde la posición actual del cursor hasta el principio de la siguiente palabra.
`d$`	Desde la posición actual del cursor hasta el final de la linea actual.
`d0`	Desde la posición actual del cursor hasta el principio de la línea.
`d^`	Desde la posición actual de cursor hasta el primer carácter no en blanco de la línea.
`dG`	Desde la línea actual al final del archivo.
`d20G`	Desde la línea actual hasta la vigésima línea del archivo.

Coloca el cursor en la palabra “It” en la primera línea de nuestro texto. Pulsa la tecla `x` repetidamente hasta que el resto de la frase se borre. A continuación, pulas la tecla `u` repetidamente hasta que se deshaga el borrado.

Nota: El auténtico vi sólo soporta un único nivel de deshacer. vim soporta múltiples niveles.

Probemos el borrado de nuevo, esta vez usando el comando `d`. De nuevo, mueve el cursor a la palabra “It” y pulsa `dW` para borrar la palabra:

```
The quick brown fox jumped over the lazy dog. was cool.

Line 2

Line 3

Line 4

Line 5
```

Pulsa `d$` para borrar desde la posición del cursor hasta el final de la línea:

```
The quick brown fox jumped over the lazy dog.
```

```
Line 2

Line 3

Line 4

Line 5
```

Pulsa `dG` para borrar desde la línea actual hasta el final del archivo:

```
~

~

~

~

~
```

Pulsa `u` tres veces para deshacer el borrado.

Cortando, copiando y pegando texto

El comando `d` no sólo borra texto, también "corta" texto. Cada vez que usamos el comando `d` el borrado se copia en un buffer de pegado (el portapapeles) al que podemos rellamar luego con el comando `p` para pegar el contenido del buffer tras el cursor o el comando `P` para pegar el contenido delante del cursor.

El comando `y` se usa para "yank" (copiar) texto de forma muy parecida a la que el comando `d` usa para cortar texto. Aquí tienes algunos ejemplos combinando el comando `y` con varios comandos de movimiento:

Tabla 13-4: Comandos de copiado

Comando	Copia
`yy`	La línea actual.
`5yy`	La línea actual y las siguientes cuatro líneas.
`yW`	Desde la posición actual del cursor hasta el principio de la siguiente palabra.
`y$`	Desde la posición actual del cursor al final de la línea actual.
`y0`	Desde la posición actual del cursor hasta el principio de la línea.
`y^`	Desde la posición actual del cursor hasta el primer espacio no en blanco de la línea.
`yG`	Desde la línea actual hasta el final de la línea.
`y20G`	Desde la línea actual hasta la vigésima línea del archivo.

Probemos algunos copiados y pegados. Coloca el cursor en la primera línea del texto y pulsa `yy` para copiar la línea actual. A continuación, mueve el cursor a la última línea (`G`) y pulsa `p` para pegar la línea debajo de la línea actual:

```
The quick brown fox jumped over the lazy dog. It was cool.

Line 2

Line 3

Line 4

Line 5
```

The quick brown fox jumped over the lazy dog. It was cool.

Tal como antes, el comando `u` deshará nuestro cambio. Con el cursor posicionado todavía en la última línea del archivo, pulsa `P` para pegar el texto sobre la línea actual:

```
The quick brown fox jumped over the lazy dog. It was cool.

Line 2

Line 3

Line 4
```

The quick brown fox jumped over the lazy dog. It was cool.

```
Line 5
```

Prueba algunos otros comandos y de la tabla anterior y trata de comprender el funcionamiento de los comandos `p` y `P`. Cuando lo hayas hecho, vuelve el archivo a su estado original.

Añadiendo líneas

`vi` es bastante estricto acerca de su idea de línea. Normalmente, no es posible mover el cursor al final de una línea y borrar el carácter de fin de línea para añadir una linea con la otra debajo. Por ésto, `vi` provee un comando específico, `J` (no confundir con `j`, que es para mover el cursor) para añadir líneas juntas.

Si colocamos el cursor en la línea 3 y escribimos el comando `J`, aquí tenemos lo que ocurre:

```
The quick brown fox jumped over the lazy dog. It was cool.

Line 2
```

Line 3 Line 4

```
Line 5
```

Buscar y reemplazar

`vi` tiene la capacidad de mover el cursor a localizaciones basadas en búsquedas. Puede hacer esto tanto en una línea individual como en un archivo completo. También puede realizar reemplazos de

texto con o sin confirmación del usuario.

Buscando dentro de una línea

El comando `f` busca en una línea y mueve el cursor a la siguiente instancia de un carácter especificado. Por ejemplo, el comando `fa` movería el cursor hasta la siguiente aparición del carácter "a" dentro de la línea actual. Después de realizar una búsqueda de un carácter en una línea, la búsqueda se repetiría escribiendo un punto y coma.

Buscando en el archivo completo

Para mover el cursor a la siguiente aparición de una palabra o frase se usa el comando `/`. Ésto funciona de la misma forma en que hemos aprendido para el programa `less`. Cuando escribes el comando `/` una "/" aparecerá en la parte de abajo de la pantalla. A continuación, escribe la palabra o frase a buscar, seguida de la tecla `Enter`. El cursor se moverá a la siguiente posición que contenga la cadena buscada. Una búsqueda puede ser repetida usando la cadena de búsqueda previa con el comando `n`. Aquí tenemos un ejemplo:

```
The quick brown fox jumped over the lazy dog. It was
cool.Line 2Line 3Line 4Line 5
```

Coloca el cursor en la primera línea del archivo. Escribe:

```
/Line
```

Seguido de la tecla `Enter`. El cursor se moverá a la línea 2. A continuación, escribe `n` y el cursor se moverá a la línea 3. Repitiendo el comando `n` moverá el cursor hacia abajo por el archivo hasta que se quede sin resultados. Aunque hasta ahora sólo hemos usado palabras y frases para nuestras búsquedas, `vi` permite el uso de *expresiones regulares*, un potente método de expresar patrones complejos de texto. Veremos las expresiones regulares en más detalle en un capítulo posterior.

Búsqueda y reemplazo global

`vi` usa un comando ex para realizar operaciones de búsqueda y reemplazo (llamadas "sustituciones" en `vi`) sobre un rango del líneas del total del archivo. Para cambiar la palabra "Line" a "line" en el archivo completo, utilizaríamos el siguiente comando:

```
:%s/Line/line/g
```

Dividamos este comando en elementos separados y veamos que hace cada uno:

Tabla 12-5: Un ejemplo de la sintaxis de la búsqueda y reemplazo global

Elemento	Significado
`:`	El carácter dos puntos comienza un comando ex.
`%`	Especifica el rango de líneas para la operación. `%` es una abreviatura que significa desde la primera a la última línea. Alternativamente, el rango podría ser especificado como `1, 5` (ya que nuestro archivo tiene cinco líneas), o `1, $` que significa "de la línea 1 a la última línea del archivo." Si el rango de líneas se omite, la operación sólo se realiza en la línea actual.
`s`	Especifica la operación. En este caso, sustitución (buscar y reemplazar).

/Line/line/	El patrón de búsqueda y el texto de reemplazo.
g	Significa "global" en sentido de que la búsqueda y reemplazo se realiza en cada instancia de la cadena de búsqueda en la línea. Si la omitimos, sólo la primera instancia de la cadena de búsqueda de cada línea se reemplazará.

Después de ejecutar nuestro comando de búsqueda y reemplazo nuestro archivo aparece así:

```
The quick brown fox jumped over the lazy dog. It was
cool.line 2line 3line 4line 5
```

Podemos especificar también un comando de sustitución con confirmación del usuario. Ésto se hace añadiendo una "c" al final del comando. Por ejemplo:

```
:%s/line/Line/gc
```

El comando cambiará nuestro archivo de nuevo a su forma previa; sin embargo, antes de cada sustitución, `vi` para y nos pregunta que confirmemos la sustitución con este mensaje:

```
replace with Line (y/n/a/q/l/^E/^Y)?
```

Cada uno de los caracteres dentro del paréntesis es una posible elección de la siguiente forma:

Tabla 12-6: Claves de confirmación de reemplazo

Clave	Acción
y	Realiza la sustitución.
n	Se salta esta instancia del patrón.
a	Realiza la sustitución en esta y todas las siguientes instancias del patrón.
q o Esc	Salir de sustituir
l	Realiza esta sustitución y luego salir. Abreviatura de "último".
Ctrl-e, Ctrl-y	Desplazamiento arriba y abajo, respectivamente. Útil para ver el contexto de la sustitución prevista.

Si escribes `y`, la sustitución se realizará, `n` hará que vi se salta esta instancia y se mueva a la siguiente.

Editando múltiples archivos

A veces es útil editar más de un archivo a la vez. Podrías necesitar hacer cambios en múltiples archivos o necesitar copiar contenido de un archivo a otro. Con `vi` podemos abrir múltiples archivos para editar especificándolos en la línea de comandos:

```
vi file1 file2 file3...
```

Salgamos de nuestra sesión de `vi` existente y creemos un nuevo archivo para editar. Escribe `:wq` para salir de `vi`, salvando nuestro texto modificado. A continuación, crearemos un archivo adicional en nuestro directorio home para que podamos jugar con él. Crearemos el archivo capturando salida del comando `ls`:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Editemos nuestro antiguo archivo y el nuevo con `vi`:

```
[me@linuxbox ~]$ vi foo.txt ls-output.txt
```

`vi` arrancará y vermos la primera línea en la pantalla:

```
The quick brown fox jumped over the lazy dog. It was
cool.Line 2Line 3Line 4Line 5
```

Cambiando entre archivos

Para cambiar de un archivo al siguiente, usaremos el este comando ex:

```
:n
```

Para volver al archivo previo usa:

```
:N
```

Como podemos movernos de un archivo a otro, `vi` impone una política que nos previene de cambiar de archivo si el archivo actual tiene cambios sin guardar. Para forzar a `vi` a cambiar de archivo y abandonar tus cambios, añade un signo de exclamación (!) al comando.

Además del método de cambio descrito antes, `vim` (y algunas versiones de `vi`) también cuenta con algunos comandos ex para hacer más fácil el manejo de múltiples archivos. Podemos ver una lista de los archivos que están siendo editados con el comando `:buffers`. Hacer esto mostrará una lista de los archivos en la parte inferior de la pantalla:

```
:buffers1 %a "foo.txt" line 12 "ls-output.txt" line 0Press
ENTER or type command to continue
```

Para cambiar a otro buffer (archivo), escribe `:buffer` seguido del número del buffer que quieres editar. Por ejemplo, para cambiar del buffer 1 que contiene el archivo `foo.txt` al buffer 2 que contiene el archivo `ls-output.txt` deberías escribir esto:

```
:buffer 2
```

y nuestra pantalla ahora muestra el segundo archivo.

Abriendo archivos adicionales para editar

También es posible añadir archivos a nuestra actual sesión de edición. El comando ex `:e` (abreviatura de "edit") seguido por el nombre del archivo abrirá el archivo adicional. Terminemos nuestra sesión de edición actual y volvamos a la línea de comandos.

Arranca `vi` de nuevo con un sólo archivo:

```
[me@linuxbox ~]$ vi foo.txt
```

Para añadir un segundo archivo, escribe:

```
:e ls-output.txt
```

Y debería aparecer en la pantalla. El primer archivo está todavía presente como podemos comprobar:

```
:buffers1 # "foo.txt" line 12 %a "ls-output.txt" line
0Press ENTER or type command to continue
```

Nota: No puedes cambiar a archivos cargados con el comando `:e` usando el comando `:n` o el comando `:N`. Para cambiar entre archivos, usa el comando `:buffer` seguido del número de buffer.

Copiando contenido de un archivo a otro

A menudo, mientras editamos múltiples archivos, querremos copiar una porción de un archivo en otro archivo que estamos editando. Ésto se hace fácilmente usando los comandos usuales de cortar y pegar que hemos usado antes. Podemos demostrarlo como sigue. Primero, usando dos archivos, cambia al buffer 1 (`foo.txt`) escribiendo:

```
:buffer 1
```

que debería darnos ésto:

```
The quick brown fox jumped over the lazy dog. It was cool.

Line 2

Line 3

Line 4

Line 5
```

A continuación, mueve el cursor al la primera línea y escribe `yy` para copiar la línea.
Cambia al segundo buffer escribiendo:

```
:buffer 2
```

La pantalla contendrá ahora listas de archivos como ésta (sólo se muestra una porción aquí):

```
total 343700

-rwxr-xr-x 1 root root 31316 2007-12-05 08:58 [

-rwxr-xr-x 1 root root 8240 2007-12-09 13:39 411toppm

-rwxr-xr-x 1 root root 111276 2008-01-31 13:36 a2p

-rwxr-xr-x 1 root root 25368 2006-10-06 20:16 a52dec

-rwxr-xr-x 1 root root 11532 2007-05-04 17:43 aafire

-rwxr-xr-x 1 root root 7292 2007-05-04 17:43 aainfo
```

Mueve el cursor a la primera línea y pega la linea copiada del archivo precedente escribiendo el comando `p`:

```
total 343700

The quick brown fox jumped over the lazy dog. It was cool.

-rwxr-xr-x 1 root root 31316 2007-12-05 08:58 [
```

```
-rwxr-xr-x 1 root root 8240 2007-12-09 13:39 411toppm

-rwxr-xr-x 1 root root 111276 2008-01-31 13:36 a2p

-rwxr-xr-x 1 root root 25368 2006-10-06 20:16 a52dec

-rwxr-xr-x 1 root root 11532 2007-05-04 17:43 aafire

-rwxr-xr-x 1 root root 7292 2007-05-04 17:43 aainfo
```

Insertando un archivo entero dentro de otro

También es posible insertar un archivo entero dentro de otro que estemos editando. Para verlo en acción, terminemos nuestra sesión `vi` y comencemos una nueva con un único archivo:

```
[me@linuxbox ~]$ vi ls-output.txt
```

Veremos nuestra lista de archivos de nuevo:

```
total 343700

-rwxr-xr-x 1 root root 31316 2007-12-05 08:58 [

-rwxr-xr-x 1 root root 8240 2007-12-09 13:39 411toppm

-rwxr-xr-x 1 root root 111276 2008-01-31 13:36 a2p

-rwxr-xr-x 1 root root 25368 2006-10-06 20:16 a52dec

-rwxr-xr-x 1 root root 11532 2007-05-04 17:43 aafire

-rwxr-xr-x 1 root root 7292 2007-05-04 17:43 aainfo
```

Movemos el cursor a la tercera línea, luego introduce el siguiente comando ex:

```
:r foo.txt
```

El comando `:r` (abreviatura de "leer") inserta el archivo especificado antes de la posición del cursor. Nuestra pantalla debería aparecer ahora así:

```
total 343700

-rwxr-xr-x 1 root root 31316 2007-12-05 08:58 [

-rwxr-xr-x 1 root root 8240 2007-12-09 13:39 411toppm

The quick brown fox jumped over the lazy dog. It was cool.

Line 2

Line 3

Line 4
```

```
Line 5

-rwxr-xr-x 1 root root 111276 2008-01-31 13:36 a2p

-rwxr-xr-x 1 root root 25368 2006-10-06 20:16 a52dec

-rwxr-xr-x 1 root root 11532 2007-05-04 17:43 aafire

-rwxr-xr-x 1 root root 7292 2007-05-04 17:43 aainfo
```

Salvando nuestro trabajo

Como todo lo demás en `vi`, hay varias formas distintas de salvar nuestros archivos editados. Ya hemos visto el comando ex `:w`, pero hay otros que podríamos encontrar útiles.

En el modo comando, escribir `ZZ` salvará el archivo actual y saldrá de `vi`. Igualmente, el comando ex `:wq` combinará los comandos `:w` y `:q` en uno que salvará el archivo y saldrá.

El comando `:w` también puede especificar un nombre de archivo opcional. Esto actúa como "Guardar como..." Por ejemplo, si estuviéramos editando `foo.txt` y quisiéramos salvar una versión alternativa llamada `foo1.txt`, escribiríamos lo siguiente:

```
:w foo1.txt
```

Nota: Mientras que el comando anterior salva el archivo bajo un nuevo nombre, no cambia el nombre del archivo que estás editando. Si continuas editando, estarás todavía editando `foo.txt`, no `foo1.txt`.

Resumiendo

Con esta serie básica de habilidades ya podemos realizar la mayoría de las ediciones de texto necesarias para mantener un sistema Linux típico. Aprender a usar `vim` en el día a día será beneficioso a largo plazo. Como los editores estilo vi están tan profundamente incluidos en la cultura Unix, veremos muchos otros programas que han estado influenciados por su diseño. `less` es un buen ejemplo de esta influencia.

Para saber más

Incluso con todo lo que hemos cubierto en este capítulo, apenas hemos arañado la superficie de lo que `vi` y `vim` pueden hacer. Aquí tienes un par de recursos en línea que puedes usar para continuar tu viaje para hacerte un maestro de `vi`:

- *Learning The vi Editor* – Un Wikilibro de Wikipedia que ofrece una guía concisa a `vi` y varios de sus similares incluyendo `vim`. Está disponible en: http://en.wikibooks.org/wiki/Vi
- *The Vim Book* – El proyecto `vim` tiene un libro de 570 páginas que cubre (casi) todas las funciones de `vim`. Puedes encontrarlo en: ftp://ftp.vim.or g/pub/vim/doc/book/vimbook-OPL.pdf .
- Un artículo de Wikipedia sobre Bill Joy, el creador de `vi`.: http://en.wikipedia.org/wiki/Bill_Joy

- Un artículo de Wikipedia sobre Bram Moolenaar, el autor de `vim`: http://en.wikipedia.org/wiki/Bram_Moolenaar

Personalizando el Prompt

En este capítulo veremos un detalle que aparentemente es trivial - nuestro prompt de shell. Al examinarlo revelaremos algunos de los trabajos internos del shell y el emulador de terminal.

Como muchas cosas en Linux, el prompt de shell es muy configurable, y aunque lo hemos dado por hecho, el prompt es un dispositivo muy útil una vez que hemos aprendido a controlarlo.

Anatomía de un prompt

Nuestro prompt por defecto es algo así:

```
[me@linuxbox ~]$
```

Fíjate que contiene nuestro nombre de usuario, el nombre de nuestro host y nuestro directorio de trabajo actual, pero ¿cómo ha tomado esta forma? Es muy simple. El prompt es definido por la variable de entorno llamada `PS1` (abreviatura de "prompt string one" - "cadena de prompt uno") Podemos ver el contenido de `PS1` con el comando `echo`:

```
[me@linuxbox ~]$ echo $PS1


[\u@\h \W]\$
```

Nota: No te preocupes si tus resultados no son exactamente los mismos que en el ejemplo. Cada distribución Linux define la cadena del prompt algo diferente, algunas algo exóticamente.

De los resultados, podemos ver que `PS1` contiene algunos caracteres que vemos en nuestro prompt como los corchetes, el signo arroba, el signo del dólar, pero el resto son misteriosos. Nuestra astucia los reconocerá como *caracteres especiales ocultos tras la barra invertida* como los que vimos en el capítulo 7. Aquí hay una lista parcial de los caracteres que el shell trata especialmente en la cadena del prompt:

Tabla 13-1: Códigos de escape usado en los prompts de shell

Secuencia	Valor mostrado
`\a`	Tono ASCII. Hace que el ordenador haga un beep cuando lo encuentre.
`\d`	Fecha actual en formato dia, mes. Por ejemplo, "Lun May 26."
`\h`	Nombre de host de la máquina local sin el nombre de dominio delante.
`\H`	Nombre de host completo.
`\j`	Número de trabajos corriendo en la sesión de shell actual.
`\l`	Nombre del dispositivo terminal actual.
`\n`	Carácter de nueva línea.
`\r`	Retorno de carro.
`\s`	Nombre del programa shell.
`\t`	Hora actual en formato 24 horas. Horas:minutos:segundos.
`\T`	Hora actual en formato 12 horas.
`\@`	Hora actual en formato 12 horas AM/PM.
`\A`	Hora actual en formato 24 horas. Horas:minutos.
`\u`	Nombre de usuario del usuario actual.
`\v`	Número de versión del shell.

\V	Números de versión y de lanzamiento del shell.
\w	Nombre del directorio de trabajo actual.
\W	Última parte del nombre del directorio de trabajo actual.
\!	Número de historial del comando actual.
\#	Número de comandos introducidos durante esta sesión de shell.
\$	Muestra un carácter “$” a menos que tengas privilegios de superusuario. En ese caso, muestra un “#” en su lugar.
\[	Señal de comienzo de una serie de uno o más caracteres no imprimibles. Se usa para agrupar caracteres de control no imprimibles que manipulará el emulador de terminal de alguna forma, como mover el cursor o cambiar colores de texto.
\]	Señales de fin de una secuencia de caracteres no imprimibles.

Probando algunos diseños de prompt alternativos

Con esta lista de caracteres especiales, podemos cambiar el prompt para ver el efecto. Primero, haremos una copia de seguridad de la cadena existente para poder recuperarla más tarde. Para hacer esto, copiaremos la cadena existente en otra variable de shell que agregaremos nosotros mismos:

```
[me@linuxbox ~]$ ps1_old="$PS1"
```

Creamos una nueva variable llamada `ps1_old` y le asignamos el valor `PS1`. Podemos verificar que la cadena ha sido copiada usando el comando `echo`:

```
[me@linuxbox ~]$ echo $ps1_old


[\u@\h \W]\$
```

Podemos recuperar el prompt original en cualquier momento de nuestra sesión de terminal simplemente revirtiendo el proceso:

```
[me@linuxbox ~]$ PS1="$ps1_old"
```

Ahora que estamos listos para seguir, veamos que ocurre si tenemos una cadena de prompt vacía:

```
[me@linuxbox ~]$ PS1=
```

Si no asignamos nada a la cadena de prompt, no tendremos nada. ¡Ninguna cadena de prompt! El prompt está ahí, pero no muestra nada, tal como le pedimos. Como así es un poco desconcertante, la reemplazaremos por un prompt minimalista:

```
PS1="\$ "
```

Así está mejor. Al menos ahora podemos ver que estamos haciendo. Fíjate el espacio en blanco delante de las comillas. Esto nos proporciona un espacio entre el signo del dólar y el cursor cuando se muestre el prompt.

Añadamos un tono a nuestro prompt:

```
$ PS1="\[\a\]\$ "
```

Ahora deberíamos escuchar un beep cada vez que se muestre el prompt. Esto podría volverse molesto, pero podría ser util ni necesitamos que nos notifique cuando un comando con un trabajo especialmente largo haya sido ejecutado. Fijate que hemos incluido las secuencias `\[` y `\]`. Como el tono ASCII (`\a`) no se “imprime”, o sea, no mueve el cursor, necesitamos decirle a bash para que

pueda determinar correctamente la longitud del prompt.

A continuación, probemos a hacer un prompt informativo con información sobre el nombre del host y lo hora:

```
$ PS1="\A \h \$ "


17:33 linuxbox $
```

Añadir la hora a nuestro prompt será útil si necesitamos seguir la pista a la hora en que se han realizado ciertas tareas. Finalmente, haremos un nuevo prompt similar al original:

```
17:37 linuxbox $ PS1="<\u@\h \W>\$ "


<me@linuxbox ~>$
```

Prueba otras secuencias de la lista anterior a ver si puedes conseguir un nuevo y brillante prompt.

Añadiendo color

La mayoría de los programas emuladores de terminal responden a ciertas secuencias de caracteres no imprimibles para controlar cosas como atributos de caracteres (color, negritas y pavorosos textos parpadeantes) y posiciones del cursor. Veremos la posición del cursor en un rato, pero primero veamos el color.

El color de los caracteres se controla enviando al emulador de terminal un *código de escape ANSI* incluido en la cadena de caracteres a mostrar. El código de control no se "imprime" en la pantalla, en su lugar es interpretado por la terminal como una instrucción. Como vimos en la tabla anterior, las secuencias `\[` y `\]` se usan para encapsular caracteres no imprimibles. Un código de escape ANSI comienza con un octal 033 (el código generado por la tecla escape), seguido de un atributo de carácter especial, seguido por una instrucción. Por ejemplo, el código para establecer el color del texto a normal (atributo = 0), texto negro es:

```
\033[0;30m
```

Aquí tenemos una tabla con los colores disponibles. Fíjate que los colores se dividen en dos grupos, diferenciados por la aplicación del carácter de atributo negrita (1) que crea la apariencia de colores "luminosos":

Tabla 17-2: Secuencias de escape usadas para establecer colores de texto

Secuencia	Color del texto	Secuencia	Color del Texto
\033[0;30m	Negro	\033[1;30m	Gris oscuro
\033[0;31m	Rojo	\033[1;31m	Rojo claro
\033[0;32m	Verde	\033[1;32m	Verde claro
\033[0;33m	Marrón	\033[1;33m	Amarillo
\033[0;34m	Azul	\033[1;34m	Azul claro

\033[0;35m	Morado	\033[1;35m	Morado claro
\033[0;36m	Cían	\033[0;36m	Cían claro
\033[0;37m	Gris claro	\033[0;37m	Blanco

Probemos un prompt rojo. Insertaremos el código de escape al principio:

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$ "

<me@linuxbox ~>$
```

Funciona, pero fíjate que todo el texto que escribamos tras el prompt también es rojo. Para arreglarlo, añadiremos otro código de escape al final del prompt que le diga al emulador de terminal que vuelve al color previo:

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$\[\033[0m\]

"

<me@linuxbox ~>$
```

¡Así está mejor!

También es posible establecer el color de fondo usando los códigos listados anteriormente. Los colores de fondo no soportan el atributo negrita.

Tabla 13-3: Secuencias de escape usadas para establecer el color de fondo

Secuencia	Color de fondo	Secuencia	Color de fondo
\033[0;40m	Negro	\033[0;44m	Azul
\033[0;41m	Rojo	\033[0;45m	Morado
\033[0;42m	Verde	\033[0;46m	Cían
\033[0;43m	Marrón	\033[0;47m	Gris claro

Podemos crear un prompt con un fondo rojo aplicando un simple cambio al primer código de escape:

```
<me@linuxbox ~>$ PS1="\[\033[0;41m\]<\u@\h \W>\$\[\033[0m\]

"

<me@linuxbox ~>$
```

Prueba los códigos de color y ¡Mira lo que puedes crear!

Nota: Además de los atributos de carácter normal (0) y negrita (1), el texto también puede tiene atributos para subrayado (4), parpadeante (5) e invertido (7). En beneficio del buen gusto, muchos emuladores de terminal rechazan el honor de tener el atributo parpadeante.

Confusión con el Terminal

Volviendo a los viejos tiempos, cuando los terminales estaban conectados a ordenadores remotos, habías muchas marcas de terminales compitiendo y todas funcionaban de forma diferente. Tenían teclados diferentes y todas tenían diferentes formas de interpretar la información de control. Los sistemas Unix y como-Unix tiene dos subsistemas complejos para tratar la torre de babel del control del terminal (llamados `termcap` y `terminfo`). Si miras en lo más profundo de la configuración de tu emulador de terminal encontrarás un parámetro para el tipo de emulador de terminal.

En un esfuerzo para hacer que los terminales hablen un tipo de lenguaje común, El American National Standards Institute (ANSI) desarrolló un catálogo de secuencias de caracteres para controlar los terminales de video. Los usuarios antiguos de DOS recordarán el archivo `ANSI.SYS` que se utilizaba para permitir la interpretación de dichos códigos.

Moviendo el cursor

Los códigos de escape pueden ser usados para posicionar el cursor. Se usa comúnmente para proporcionar un reloj o algún otro tipo de información en una localización diferente de la pantalla, como en la esquina superior cada vez que se dibuja el prompt. Aquí tenemos una lista de los códigos de escape que posicionan el cursor:

Tabla 13-4: Secuencias de escape de movimiento del cursor

Código de escape	Acción
`\033[1,cH`	Mueve el cursor a la línea *l* y columna *c*
`\033[nA`	Mueve el cursor arriba *n* lineas
`\033[nB`	Mueve el cursor abajo *n* lineas
`\033[nC`	Mueve el cursor hacia delante *n* caracteres
`\033[nD`	Mueve el cursor hacia atrás *n* caracteres
`\033[2J`	Limpia la pantalla y mueve el cursor a la esquina superior izquierda (línea 0, columna 0)
`\033[K`	Limpia desde la posición del cursor hasta el final de la línea actual
`\033[s`	Almacena la posición actual del cursor
`\033[u`	Recupera la posición del cursor almacenada

Usando los códigos anteriores, construiremos un prompt que dibuje una línea roja arriba de la pantalla conteniendo un reloj (dibujado en texto amarillo) cada vez que se muestre el prompt. El código para el prompt es esta cadena de aspecto formidable:

```
PS1="\
[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]
<\u@\h \W>\$ "
```

Echemos un vistazo a cada parte de la cadena para ver que hace:

Tabla 13-5: Desglose de cadenas de prompt complejas

Secuencia	Acción
`\[`	Comienza una secuencia de caracteres no imprimibles. El propósito de ésto es permitir que bash calcule apropiadamente el tamaño del prompt visible. Sin un calculo preciso, las funciones de edición de la línea de comandos no pueden posicionar el cursor apropiadamente.

\033[s	Almacena la posición del cursor. Se necesita para volver a la localización del prompt después de que la barra y reloj hayan sido dibujados en la parte superior de la pantalla. *Ten en cuenta que ciertos emuladores de terminal no soportan este código*.
\033[0;0H	Mueve el cursor a la esquina superior izquierda, que es la línea 0, columna 0.
\033[0;41m	Establece el color de fondo a rojo.
\033[K	Borra la posición actual del cursor (la esquina superior izquierda) al final de la línea. Como el color de fondo es ahora rojo, la línea se borra a ese color creando nuestra barra. Nota que borrar hasta el final de la línea no cambia la posición del cursor, que permanece en la esquina superior izquierda.
\033[1;33m	Establece el color a amarillo
\t	Muestra la hora actual. Como es un elemento "imprimible", lo incluimos todavía dentro de la porción no imprimible del prompt, ya que no queremos que bash incluya el reloj cuando calcule el tamaño real del prompt mostrado.
\033[0m	Apaga el color. Afecta tanto al texto como al fondo.
\033[u	Recupera la posición del cursor salvado anteriormente.
\]	Fin de la secuencia de caracteres no imprimibles.
<\u@\h \W>\$	Cadena de prompt.

Salvando del prompt

Obviamente, no queremos escribir este monstruo todo el tiempo, así que querremos almacenar nuestro prompt en alguna parte. Podemos hacer el prompt permanente añadiéndolo a nuestro archivo .bashrc. Para hacerlo, añade estas dos líneas al archivo:

```
PS1="\
[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]
<\u@\h \W>\$ "

export PS1
```

Resumiendo

Lo creas o no, hay mucho más que puede hacerse con los prompts incluyendo funciones de shell y scripts que no hemos visto aquí, pero es un buen comienzo. No todo el mundo va a preocuparse de cambiar el prompt, ya que el prompt por defecto normalmente es satisfactorio. Pero para aquellos de nosotros a los que nos gusta juguetear, el shell nos brinda la oportunidad de pasar muchas horas de diversión trivial.

Para saber más

- El HOWTO del Bash Prompt del Linux Documentation Project proporciona una muy completa exposición sobre para que esta hecho el prompt de shell. Está disponible en: http://tldp.org/HOWTO/Bash-Prompt-HOWTO/
- Wikipedia tiene un buen artículo sobre los códigos de escape ANSI:

Codes:
http://en.wikipedia.org/wiki/ANSI_escape_code

Gestión de paquetes

Si pasamos algún tiempo en la comunidad Linux, oímos muchas opciones acerca de que distribución es la “mejor”. A menudo, estas discusiones se vuelven muy tontas, centrándose en cosas como la belleza del fondo de pantalla (¡Algunas personas no usan Ubuntu por su gama de colores por defecto!) y otros asuntos sin importancia.

El aspecto más importante que determina la calidad de una distribución es el *sistema de paquetes* y la vitalidad de la comunidad que soporte la distribución. Según pasemos más tiempo con Linux, veremos que su ecosistema de software es extremadamente dinámico. Las cosas cambian constantemente. La mayoría de las distribuciones Linux líderes liberan nuevas versiones cada seis meses y muchos programas individuales se actualizan cada día. Para enfrentarse a esta ventisca de software, necesitamos buenas herramientas para la *gestión de paquetes*.

La gestión de paquetes es un método para instalar y mantener software en el sistema. Hoy, la mayoría de la gente puede satisfacer todas sus necesidades de software instalando *paquetes* de su distribuidor Linux. Ésto contrasta con los primeros días de Linux, cuando uno tenía que descargar y compilar código fuente para instalar software. No es que haya nada malo en compilar *código fuente*; de hecho, tener acceso al código fuente es una de las grandes maravillas de Linux. Nos da (y a todos los demás también) la capacidad de examinar y mejorar el sistema. Lo único es que tener un paquete precompilado es más rápido y fácil de manejar.

En este capítulo, veremos algunas herramientas de línea de comandos que se usan para manejar los paquetes. Aunque todas las distribuciones importantes ofrecen programas gráficos potentes y sofisticados para mantener el sistema, es importante aprender programas de la línea de comandos también. Pueden realizar muchas tareas que son difíciles (o imposibles) de hacer con sus homólogos gráficos.

Sistemas de paquetes

Diferentes distribuciones usan diferentes sistemas de paquetes y como regla general, un paquete destinado a una distribución no es compatible con otra distribución. La mayoría de las distribuciones se clasifican dentro de dos familias de tecnologías de paquetes: la familia Debian “.deb” y la familia Red Hat “.rpm”. Hay algunas excepciones importantes como Gentoo, Slackware y Foresight, pero la mayoría de las demás usan uno de estos dos sistemas básicos.

Tabla 14-1: Principales familias de sistemas de paquetes

Sistema de paquetes	Distribuciones (Lista parcial)
Estilo Debian (.deb)	Debian, Ubuntu, Xandros, Linspire
Estilo Red Hat (.rpm)	Fedora, CentOS, Red Hat Enterprise Linux, OpenSUSE, Mandriva, PCLinuxOS

Cómo funciona un sistema de paquetes

El método de distribución de software utilizado en la industria del software propietario normalmente implica comprar un tipo de medio de instalación como un “disco de instalación” y luego ejecutar un “asistente de instalación” para instalar una nueva aplicación en el sistema.

Linux no funciona de esa forma. Virtualmente todo el software para un sistema Linux se encuentra

en Internet. La mayoría lo proveerá el suministrador de la distribución en forma de *paquetes* y el resto estará disponible en código fuente para ser instalado manualmente. Hablaremos un poco sobre cómo instalar software compilando el código fuente en un capítulo posterior.

Paquetes

La unidad básica de software en los sistemas de paquetes es el *paquete*. Un paquete es una colección comprimida de archivos que incluyen el paquete de software. Un paquete puede consistir en numerosos programas y archivos de datos que soportan el programa. Además de los archivos a instalar, el paquete también incluye metadatos sobre el paquete, como una descripción en texto del paquete y su contenido. Adicionalmente, muchos paquetes contienen scripts de pre y post instalación que realizan tareas de configuración antes y después de la instalación del paquete.

Los paquetes son creados por una persona conocida como el *mantenedor del paquete*, a menudo (pero no siempre) un empleado del suministrador de la distribución. El mantenedor del paquete toma el software en código fuente del *proveedor superior* (el autor del programa), lo compila, y crea los metadatos del paquete y todos los scripts de instalación necesarios. A menudo, el mantenedor del paquete aplicará modificaciones al código fuente original para mejorar la integración del programa con otras partes de la distribución Linux.

Repositorios

Mientras que algunos proyectos de software eligen realizar su propia distribución y empaquetado, la mayoría de los paquetes hoy son creados por los suministradores de la distribución y terceras partes interesadas. Los paquetes son puestos a disposición de los usuarios de una distribución en repositorios centrales que pueden contener miles de paquetes, cada uno construidos y mantenidos especialmente para la distribución.

Una distribución puede mantener varios repositorios diferentes para fases diferentes del ciclo de desarrollo del software. Por ejemplo, normalmente habrá un repositorio de “pruebas” que contendrá paquetes que acaban de ser construidos y que están destinados a almas valientes que buscan errores antes de que sean liberados para distribución general. Un distribución, a menudo, tiene un repositorio de “desarrollo” donde se guardan los paquetes en proceso de trabajo para incluirlos en la siguiente versión principal de la distribución.

Una distribución también puede tener repositorios de terceros relacionados. Éstos a menudo se necesitan para proporcionar software que, por razones legales como patentes o problemas DRM anti piratería, no pueden ser incluidos con la distribución. Quizá el caso más conocido es el del soporte del encriptado de los DVD, que no es legal en Estados Unidos. Los repositorios de terceros funcionan en países donde las patentes de software y las leyes anti pirateria no se aplican. Estos repositorios son normalmente independientes de la distribución que soportan y para usarlos, debemos conocerlos y incluirlos manualmente en los archivos de configuración del sistema de gestión de paquetes.

Dependencias

Los programas raras veces son “autónomos”; en su lugar necesitan la presencia de otros componentes de software para hacer su trabajo. Actividades comunes, como entrada/salida por ejemplo, están soportadas por rutinas compartidas por muchos programas. Estas rutinas se almacenan en lo que llamamos *librerías compartidas*, que proporcionan servicios esenciales a más de un programa. Si un paquete requiere un recurso compartido como una librería compartida, se dice que tiene *dependencia*. Los sistemas de gestión de paquetes modernos cuentan con un método

de *resolución de dependencias* para asegurarse de que cuando un paquete se instala, todas sus dependencias se instalan también.

Herramientas de paquetes de alto y bajo nivel

Los sistemas de gestión de paquetes normalmente consisten en dos tipos de herramientas: herramientas de bajo nivel que soportan tareas como instalar y eliminar paquetes, y herramientas de alto nivel que realizan búsquedas por metadatos y resolución de dependencias. En este capítulo, veremos las herramientas proporcionadas por los sistemas estilo Debian (como Ubuntu y muchos otros) y aquellas usadas por productos Red Hat recientes. Mientras que todas las distribuciones estilo Red Hat coinciden en el mismo programa de bajo nivel (`rpm`), usan distintas herramientas de alto nivel. Para nuestro tema, veremos el programa de alto nivel `yum`, usado por Fedora, Red Hat Enterprise Linux y CentOS. Otras distribuciones estilo Red Hat cuentan con herramientas de alto nivel con características parecidas.

Tabla 14-2: Herramientas de sistemas de paquetes

Distribuciones	Herramientas de bajo nivel	Herramientas de alto nivel
Estilo Debian	`dpkg`	`apt-get, aptitude`
Fedora, Red Hat Enterprise Linux, CentOS	`rpm`	`yum`

Tareas comunes en la gestión de paquetes

Hay muchas operaciones que pueden ser realizadas con las herramientas de gestión de paquetes de la línea de comandos. Veremos las más comunes. Ten en cuenta que las herramientas de bajo nivel también soportan la creación de paquetes, una actividad fuera del objetivo de este libro.

En el siguiente apartado, el término “*nombre_paquete”* ser refiere al nombre real del paquete en lugar del término “*archivo_paquete”*, que es el nombre del archivo que contiene el paquete.

Buscando un paquete en un repositorio

Usando las herramientas de alto nivel para buscar metadatos en un repositorio, un paquete puede ser localizado basándonos en su nombre o descripción.

Tabla 14-3: Comandos de búsqueda de paquetes

Estilo	Comando(s)
Debian	`apt-get update` `apt-cache search` *cadena_buscada*
Red Hat	`yum search` *cadena_buscada*

Ejemplo: Para buscar el editor de texto emacs en un repositorio yum, podemos usar este comando:

```
yum search emacs
```

Instalando un paquete de un repositorio

Las herramientas de alto nivel permiten descargar e instalar un paquete de un repositorio con resolución completa de dependencias.

Tabla 14-4: Comandos de instalación de paquetes

Estilo	Comando(s)
Debian	`apt-get update` `apt-get install` *nombre_paquete*
Red Hat	`yum install` *nombre_paquete*

Ejemplo: Para instalar el editor de texto emacs de un repositorio apt:

```
apt-get update; apt-get install emacs
```

Instalando un paquete de un archivo de paquetes

Si un archivo de paquetes ha sido descargado de una fuente distinta a un repositorio, puede ser instalado directamente (aunque sin resolución de dependencias) usando una herramienta de bajo nivel:

Tabla 14-5: Comandos de instalación de paquetes de bajo nivel

Estilo	Comando(s)
Debian	`dpkg --install` *archivo_paquete*
Red Hat	`rpm -i` *archivo_paquete*

Ejemplo: si el archivo de paquetes `emacs-22.1-7.fc7-i386` ha sido descargado de un sitio que no sea un repositorio, podría ser instalado de la siguiente forma:

```
rpm -i emacs-22.1-7.fc7-i386.rpm
```

Nota: Como esta técnica usa el programa de bajo nivel `rpm` para realizar la instalación, no se realiza ninguna resolución de dependencias. Si `rpm` descubre una dependencia que falta, `rpm` terminará con un error.

Eliminando un paquete

Los paquetes pueden ser desinstalados usando tanto las herramientas de bajo nivel como de alto nivel. Las herramientas de alto nivel se muestran a continuación.

Tabla 15-6: Comandos de eliminación de paquetes

Estilo	Comando(s)
Debian	`apt-get remove` *nombre_paquete*
Red Hat	`yum erase` *nombre_paquete*

Ejemplo: para desinstalar el paquete emacs de un sistema estilo Debian:

```
apt-get remove emacs
```

Actualizando paquetes de un repositorio

La tarea de gestión de paquetes más común es mantener el sistema actualizado con los últimos paquetes. Las herramientas de alto nivel pueden realizar estas tareas vitales en un paso único.

Tabla 14-7: Comandos de actualización de paquetes

Estilo	Comando(s)

Debian	`apt-get update; apt-get upgrade`
Red Hat	`yum update`

Ejemplo: para aplicar todas las actualizaciones disponibles a los paquetes instalados en un sistema estilo Debian:

```
apt-get update; apt-get upgrade
```

Actualizando un paquete desde un archivo de paquetes

Si una versión actualizada de un paquete ha sido descargada desde una fuente que no es un repositorio, puede ser instalada reemplazando a la anterior versión:

Tabla 14-8: Comandos de actualización de paquetes de bajo nivel

Estilo	Comando(s)
Debian	`dpkg --install` *archivo_paquete*
Red Hat	`rpm -U` *archivo_paquete*

Ejemplo: Actualizar una instalación existente de emacs a la versión contenida en el archivo de paquetes `emacs-22.1-7.fc7-i386.rpm` en un sistema Red Hat:

```
rpm -U emacs-22.1-7.fc7-i386.rpm
```

Nota: `dpkg` no tiene una opción específica para actualizar un paquete en lugar de instalarlo tal como lo hace `rpm`.

Listando los paquetes instalados

Estos comandos pueden utilizarse para mostrar una lista de todos los paquetes instalados en el sistema:

Tabla 14-9: Comandos de listado de paquetes

Estilo	Comando(s)
Debian	`dpkg --list`
Red Hat	`rpm -qa`

Determinando si un paquete está instalado

Estas herramientas de bajo nivel pueden utilizarse para mostrar si un paquete específico está instalado:

Tabla 14-10: Comandos de estado de paquetes

Estilo	Comando(s)
Debian	dpkg –status *nombre_paquete*
Red Hat	rpm -q *nombre_paquete*

Ejemplo: Para determinar si el paquete emacs está instalado en un sistema estilo Debian:

```
dpkg –status emacs
```

Mostrando información acerca de un paquete instalado

Si sabemos el nombre de un paquete instalado, los siguientes comandos pueden utilizarse para mostrar una descripción del paquete:

Tabla 14-11: Comandos de información de paquetes

Estilo	Comando(s)
Debian	apt-cache show *nombre_paquete*
Red Hat	yum info *nombre_paquete*

Ejemplo: Para ver una descripción del paquete emacs en un sistema estilo Debian:

```
apt-cache show emacs
```

Buscando qué paquete ha instalado un archivo

Para determinar que paquete es responsable de la instalación de un archivo en particular, podemos usar los siguientes comandos:

Tabla 14-12: Comandos de identificación del archivo de paquetes

Estilo	Comando(s)
Debian	dpkg --search archivo_*paquete*
Red Hat	rpm -qf archivo_*paquete*

Ejemplo: Para ver que paquete instaló el archivo `/usr/bin/vim` en un sistema Red Hat:

```
rpm -qf /usr/bin/vim
```

Resumiendo

En los siguientes capítulos, exploraremos muchos programas diferentes cubriendo un amplio rango

de áreas de aplicaciones. Mientras que la mayoría de los programas se instalan normalmente por defecto, necesitaremos instalar paquetes adicionales si programas necesarios no están instalados en nuestro sistema. Con nuestro conocimiento recién obtenido (y valorado) de gestión de paquetes, no deberíamos tener problemas instalando y manejando los programas que necesitemos.

El mito de la instalación de software Linux

La gente que migra desde otras plataformas, a veces, son víctimas del mito de que el software es algo difícil de instalar en Linux y que la variedad de sistemas de paquetes usados por las diferentes distribuciones es un impedimento. Bien, es un impedimento, pero sólo para los vendedores de software propietario que deseen distribuir versiones binarias de su software secreto.

El ecosistema Linux se basa en la idea del código abierto. Si el desarrollador de un programa libera el código fuente de un producto, es probable que una persona asociada a la distribución empaquete el producto y lo incluya en su repositorio. Este método asegura que el producto se integra bien con la distribución y que el usuario tiene la comodidad de una "única tienda", en lugar de tener que buscar la web de cada producto.

Los controladores de dispositivos son manejados de forma muy parecida, excepto que en lugar de ser elementos separados en el repositorio de una distribución, forman parte del propio kernel Linux. Hablando en general, no hay algo como un "disco de controladores" en Linux. O el kernel soporta un dispositivo o no lo hace, y el kernel Linux soporta muchos dispositivos. Muchos más, de hecho que Windows. De acuerdo, esto no consuela si el dispositivo en concreto que necesitas no está soportado. Cuando esto ocurre, necesitas ver la causa. Una falta de soporte de controladores está a menudo causada por una de estas tres cosas:

1. El dispositivo es demasiado nuevo. Como muchos distribuidores de hardware no soportan activamente el desarrollo Linux, falta, a menos que un miembro de la comunidad Linux escriba el código del controlador para el kernel. Esto lleva tiempo.

2. El dispositivo es demasiado exótico. No todas las distribuciones incluyen cada controlador de dispositivo posible. Cada distribución desarrolla sus propios kernels, y como los kernels son muy configurables (lo que hace posible ejecutar Linux en todo, desde relojes de pulsera a grandes servidores) podrían haber pasado por alto un dispositivo en particular. Para encontrar y descargar el código fuente del controlador, puedes (sí tú) compilar e instalar el controlador tú mismo. Este proceso no es demasiado difícil, pero sí que es enredado. Hablaremos sobre compilar software en un capítulo posterior.

3. El vendedor de hardware esconde algo. Ni ha liberado el código fuente para el controlador de Linux, y ni ha liberado la información técnica para que alguien lo cree. Esto significa que el proveedor de hardware está tratando de mantener las interfaces de programación al dispositivo en secreto. Como no queremos dispositivos secretos en nuestros ordenadores, sugiero que quitemos el hardware ofensivo y lo tiremos a la basura con otras cosas inútiles.

Para saber más

Echa un rato conociendo el sistema de gestión de paquetes de tu distribución. Cada distribución proporciona documentación para sus herramientas de gestión de paquetes. Además, aquí hay algunas fuentes genéricas más:

- El capítulo de gestión de paquetes del FAQ de Debian GNU/Linux, proporciona una vista general de los sistemas de paquetes en sistemas Debian: http://www.debian.org/doc/FAQ/ch-pkgtools.en.html
- La página principal del proyecto RPM: http://www.rpm.org
- La página principal del proyecto YUM en la Universidad Duke: http://linux.duke.edu/projects/yum/
- Para algo más a fondo, la Wikipedia tiene un artículo sobre metadatos: http://en.wikipedia.org/wiki/Metadata

Medios de almacenamiento

En capítulos anteriores hemos visto como manejar datos a nivel de archivos. En este capítulo, consideraremos los datos a nivel de dispositivo. Linux tiene capacidades sorprendentes para soportar dispositivos de almacenamiento, desde almacenamiento físico, como discos duros, a dispositivos de almacenamiento virtuales como RAID (Redundant Array of Independent Disk - Conjunto redundante de discos independientes) y LVM (Logical Volume Manager – Administrador de volúmenes lógicos).

Sin embargo, como esto no es un libro sobre administración de sistemas, no trataremos de cubrir este aspecto en profundidad. Lo que trataremos de hacer es presentar algunos de los conceptos y comandos que se usan para manejar dispositivos de almacenamiento.

Para realizar los ejercicios de este capítulo, usaremos una unidad flash USB, un disco CD-RW (para sistemas equipados con un grabador de CD-ROM) y un disquete (de nuevo, si el sistema está equipado).

Veremos los siguientes comandos:

- `mount` – Monta un sistema de ficheros
- `umount` – Desmonta un sistema de ficheros
- `fsck` – Chequea y repara un sistema de ficheros
- `fdisk` – Manipulador de tablas de particiones
- `mkfs` – Crea un sistema de ficheros
- `fdformat` – Formatea un disquete
- `dd` – Escribe datos orientados a bloques directamente en un dispositivos
- `genisoimage (mkisofs)` – Crea un archivo de imagen ISO 9660
- `wodim (cdrecord)` – Escribe datos en una unidad de almacenamiento óptico

- `md5sum` – Calcula un checksum MD5

Montando y desmontando dispositivos de almacenamiento

Avances recientes en el escritorio de Linux han hecho la gestión de los dispositivos de almacenamiento extremadamente sencillo para los usuarios de escritorio. La mayoría de las veces, conectamos un dispositivo a nuestro sistema y "simplemente funciona". Volviendo a los viejos tiempos (digamos, 2004), ésto había que hacerlo manualmente. En sistemas sin escritorio (p.ej.: Servidores) esto sigue siendo en gran parte un procedimiento manual ya que los servidores tienen, a menudo, almacenamientos extremos que necesitan configuraciones complejas.

El primer paso en la gestión de un dispositivo de almacenamiento es conectar el dispositivo en el árbol de sistema de archivos. Este proceso, llamado *montaje*, permite al dispositivo participar en el sistema operativo. Como vimos en el Capítulo 2, los sistemas operativos como-Unix, por ejemplo Linux, mantienen un único árbol de sistema de archivos con los dispositivos conectados en varios puntos. Ésto contrasta con otros sistemas operativos como MS-DOS y Windows que mantienen árboles de sistemas de archivos separados para cada dispositivo (por ejemplo `C:\`, `D:\`, etc.).

Un archivo llamado `/etc/fstab` lista los dispositivos (típicamente particiones de disco duro) que se montan en el arranque del sistema. Aquí tenemos un ejemplo de archivo `/etc/fstab` de un sistema Fedora 7:

```
LABEL=/12       /         ext3   defaults        1 1

LABEL=/home     /home     ext3   defaults        1 2

LABEL=/boot     /boot     ext3   defaults        1 2

tmpfs           /dev/shm  tmpfs  defaults        0 0

devpts          /dev/pts  devpts gid=5,mode=620  0 0

sysfs           /sys      sysfs  defaults        0 0

proc            /proc     proc   defaults        0 0

LABEL=SWAP-sda3 swap      swap   defaults        0 0
```

La mayoría de los sistemas de archivos listados en el ejemplo son virtuales y no se aplican a nuestro tema. Para nuestros propósitos, los interesantes son los tres primeros:

```
LABEL=/12   /     ext3 defaults 1 1

LABEL=/home /home ext3 defaults 1 2

LABEL=/boot /boot ext3 defaults 1 2
```

Son las particiones del disco duro. Cada línea del archivo consiste en seis campos, como sigue:

Tabla 15-1:Campos de /etc/fstab

Campo	Contenido	Descripción
1	Dispositivo	Tradicionalmente, este campo contiene el nombre real de un archivo de dispositivo asociado con el dispositivo físico, como `/dev/hda1` (la primera partición del dispositivo maestro en el primer canal IDE). Pero con los ordenadores de hoy, que tienen muchos dispositivos que son plug & play (como unidades USB), muchas distribuciones Linux modernas asocian un dispositivo con una etiqueta de texto en su lugar. Esta etiqueta (que se añade al medio de almacenamiento cuando se formatea) la lee el sistema operativo cuando el dispositivo es conectado al sistema. De esta forma, no importa que archivo de dispositivo se le asigna al dispositivo físico real, puede permanecer correctamente indefinido.
2	Punto de montaje	El directorio donde el dispositivo es conectado dentro del árbol del sistema de archivos.
3	Tipo de sistema de archivos	Linux permite montar muchos tipos de sistemas de archivos. La mayoría de los sistemas de archivos nativos en Linux son `ext3`, pero soporta muchos más, como `FAT16 (msdos), FAT32 (vfat), NTFS (ntfs), CD-ROM (iso9660),` etc.
4	Opciones	Los sistemas de archivos pueden montarse con varias opciones. Es posible, por ejemplo, montar sistemas de archivos como sólo-lectura, o prevenir que los programas se ejecuten desde él (una característica de seguridad útil para medios movibles).
5	Frecuencia	Un único número que especifica sí y cuando un sistema de archivos va a realizar una copia de seguridad con el comando `dump`.
6	Orden	Un único número que especifica en qué orden va a ser chequeado el sistema de archivos con el comando `fsck`.

Viendo una lista de los sistemas de archivos montados

El comando `mount` se usa para montar sistemas de archivos. Introducir el comando sin argumentos mostrará una lista de los sistemas de archivos actualmente montados:

```
[me@linuxbox ~]$ mount
/dev/sda2 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda5 on /home type ext3 (rw)
/dev/sda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
```

```
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
/dev/sdd1 on /media/disk type vfat (rw,nosuid,nodev,noatime,
uhelper=hal,uid=500,utf8,shortname=lower)
twin4:/musicbox on /misc/musicbox type nfs4 (rw,addr=192.168.1.4)
```

El formato del listado es: *dispositivo on punto_de_montaje type tipo_sistema_archivos (opciones)*.

Por ejemplo, la primera línea muestra que el dispositivo `/dev/sda2` está montado como la raíz del sistema de archivos, en tipo ext3, y que es tanto legible como modificable (la opción “rw”). Este listado también tiene dos entradas interesantes al final de la lista. La ante penúltima entrada muestra una tarjeta de memoria SD de 2 gibabytes montada en `/media/disk`, y la última entrada es un disco de red montado en `/misc/musicbox`.

Para nuestro primer experimento, trabajaremos con un CD-ROM. Primero, veamos el sistema antes de insertar el CD-ROM:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
```

El listado es de un sistema CentOS 5, que usa LVM (Logical Volume Manager) para crear su sistema de archivos raíz. Como muchas distribuciones Linux modernas, este sistema tratará de montar automáticamente el CD-ROM después de ser insertado. Después de insertar el disco, veremos lo siguiente:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/hdc on /media/live-1.0.10-8 type iso9660 (ro,noexec,nosuid,
nodev,uid=500)
```

Tras insertar el disco, vemos la misma lista de antes con una entrada adicional. Al final de la lista vemos que el CD-ROM (que es el dispositivo `/dev/hdc` en este sistema) ha sido montado en `/media/live-1.0.10-8`, y es tipo iso9660 (un CD-ROM). Para el propósito de nuestro experimento, nos interesa el nombre del dispositivo. Cuando hagas este experimento tú mismo, el nombre del dispositivo será probablemente distinto.

Advertencia: En los ejemplos siguientes, es de vital importancia que prestes mucha atención a los nombres reales de los dispositivos en uso en tu sistema y **¡no usar los nombres utilizados en este texto!**

Ten en cuenta también que los CDs de audio no son lo mismo que los CD-ROMs. Los CDs de audio no contienen sistemas de archivos y por lo tanto no pueden ser montados en la forma normal.

Ahora que tenemos el nombre de dispositivo de la unidad de CD-ROM, desmontemos el disco y lo montaremos de nuevo en otra localización en el árbol del sistema de archivos. Para hacerlo, nos haremos superusuario (usando el comando apropiado para nuestro sistema) y desmontamos el disco con el comando `umount` (fíjate en el deletreo):

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]# umount /dev/hdc
```

El siguiente paso es crear un *nuevo punto de montaje* para el disco. Un punto de montaje es simplemente un directorio en algún lugar del árbol del sistema de archivos. No tiene nada de especial. No tiene por qué ser un directorio vacío, pero si montas un dispositivo en un directorio no vacío, no podrás ver el contenido previo del directorio a no ser que desmontes el dispositivo. Para nuestro propósito, crearemos un nuevo directorio:

```
[root@linuxbox ~]# mkdir /mnt/cdrom
```

Finalmente, montamos el CD-ROM en un nuevo punto de montaje. La opción `-t` se usa para especificar el tipo de sistema de archivos:

```
[root@linuxbox ~]# mount -t iso9660 /dev/hdc /mnt/cdrom
```

Después de ésto, podemos examinar el contenido del CD-ROM via el nuevo punto de montaje:

```
[root@linuxbox ~]# cd /mnt/cdrom
[root@linuxbox cdrom]# ls
```

Fíjate que ocurre cuando tratarmos de desmontar el CD-ROM:

```
[root@linuxbox cdrom]# umount /dev/hdc
umount: /mnt/cdrom: device is busy
```

¿Por qué pasa ésto? La razón es que no podemos desmontar un dispositivo si está siendo usado por alguien o por algún proceso. En este caso, hemos cambiado nuestro directorio de trabajo al punto de montaje del CD-ROM, lo que causa que el dispositivo esté ocupado. Podemos remediar el problema fácilmente cambiando el directorio de trabajo a otro que no sea el punto de montaje:

```
[root@linuxbox cdrom]# cd
[root@linuxbox ~]# umount /dev/hdc
```

Ahora el dispositivo se desmonta con éxito.

Por qué es importante desmontar

Si vemos la salida del comando `free`, que muestra estadísticas sobre el uso de memoria, verás una estadística llamada "buffers". Los sistemas de cómputo está diseñados para ir lo más rápido posible. Uno de los impedimentos a la velocidad del sistema son los dispositivos lentos. Las impresoras son un buen ejemplo. Incluso la impresora más rápida es extremadamente lenta para los estándares de los ordenadores. Un ordenador sería muy lento si tuviera que parar y esperar a que la impresora termine de imprimir una página. En los primeros días de los Pcs (antes de la multitarea), esto era un problema real. Si estabas trabajando en una hoja de cálculo o un documento de texto, el ordenador se paraba y se volvía indisponible cada vez que imprimías. El ordenador enviaba los datos a la impresora tan rápido como la impresora podía aceptarlos, pero era muy lento ya que las impresoras no imprimen muy rápido. Éste problema se solucionó con la llegada del *buffer de impresora*, un dispositivo con memoria RAM que podía situarse entre el ordenador y la impresora. Con el buffer de impresora instalado, el ordenador podía enviar la salida de la impresora al buffer y era rápidamente almacenado en la rápida RAM para que el ordenador pudiera volver al trabajo sin esperar. Mientras tanto, el buffer de impresora iba enviando la *cola* de datos a la impresora lentamente desde la memoria del buffer a la velocidad a que la impresora podía aceptarlos.

La idea del buffer se usa abundantemente en ordenadores para hacerlos más rápidos. Evitan que ocasionalmente leer o escribir datos de o hacia dispositivos lentos impidan la velocidad del sistema. Los sistemas operativos almacenan en memoria datos que han sido leídos, y tienen que ser escritos en dispositivos de almacenamiento por el máximo tiempo posible antes que tener que interactuar con el dispositivo lento. En un sistema Linux por ejemplo, notarás que el sistema parece tener la memoria llena cuanto más tiempo se usa. Esto no significa que Linux esté "utilizando" toda la memoria, significa que Linux está sacando beneficio de toda la memoria disponible para hacer todo el buffering que pueda.

Este buffer permite que escribir en dispositivos de almacenamiento se haga rápidamente, porque la escritura en el dispositivo físico ha sido pospuesto a un tiempo futuro. Al mismo tiempo, los datos destinados al dispositivo se van acumulando en la memoria. De vez en cuando, el sistema operativo escribirá estos datos en el dispositivo físico.

Desmontar un dispositivo implica escribir todos los datos restantes en el dispositivo para que pueda ser retirado con seguridad. Si se retira el dispositivo sin desmontarlo primero, existe la posibilidad que no todos los datos destinados al dispositivo hallan sido transferidos. En algunos casos, estos datos pueden incluir actualizaciones vitales de directorios, que causará una *corrupción del sistema de archivos*, una de las peores cosas que le puede ocurrir a un ordenador.

Determinando los nombres de los dispositivos

A veces es difícil determinar el nombre de un dispositivo. En los viejos tiempos, no era muy difícil. Un dispositivo siempre estaba en el mismo sitio y no cambiaba. Los sistemas como-Unix siguen así. Volviendo a cuando Unix se desarrolló, "cambiar una unidad de disco" implicaba usar un montacargas para retirar un dispositivo del tamaño de una lavadora de la habitación del ordenador. En tiempos recientes, la configuración del hardware de escritorio se ha vuelto algo más dinámica y Linux ha evolucionado para hacerse más flexible que sus antepasados.

En los ejemplos siguientes sacaremos ventaja de la capacidad de los escritorios modernos de Linux para montar los dispositivos "automágicamente" y determinar el nombre después. Pero ¿qué pasa si estamos administrando un servidor o algún otro entorno donde esto no ocurre? ¿cómo podemos resolverlo?

Primero, veamos cómo el sistema nombra los dispositivos. Si listamos el contenido del directorio `/dev` (donde viven todos los dispositivos), podemos ver que hay montones y montones de dispositivos:

```
[me@linuxbox ~]$ ls /dev
```

El contenido del listado revela algunos patrones en el nombrado de los dispositivos. Aquí tenemos unos pocos:

Tabla 15-2: Nombres de los dispositivos de almacenamiento en Linux

Patrón	Dispositivo
`/dev/fd*`	Disquetes
`/dev/hd*`	Discos IDE (PATA) en sistemas antiguos. Las placas báse típicas contienen dos conectores IDE o *canales*, cada una con un cable con dos puntos de conexión para unidades. La primera unidad del cable se llama dispositivo *maestro* y la segunda se llama dispositivo *esclavo*. Los nombres de dispositivos se ordenan de tal forma que `/dev/hda` se refiere al dispositivo maestro del primer canal, `/dev/hdb` es el dispositivo esclavo del primer canal; `/dev/hdc`, el dispositivo maestro del segundo canal, etc. Un dígito al final indica el número de partición en el dispositivo. Por ejemplo, `/dev/hda1` se refiere a la primera partición del primer disco duro del sistema mientras que `/dev/hda` se refiere al disco completo.
`/dev/lp*`	Impresoras
`/dev/sd*`	Discos SCSI. En sistemas Linux recientes, el kernel trata todos los dispositivos

	que sirven de disco (incluyendo discos duros PATA/SATA, unidades flash, dispositivos de almacenamiento USB como reproductores de música portátiles y cámaras digitales) como discos SCSI. El resto del sistema de nombres es similar al antiguo esquema de nombres /dev/hd* descrito arriba.
/dev/sr*	Discos ópticos (lectores y grabadores de CD/DVD).

Además, vemos a menudo enlaces simbólicos como /dev/cdrom, /dev/dvd y /dev/floppy que apuntan a los archivos de los dispositivos reales, ofrecidos como un servicio.

Si estás trabajando en un sistema que no monta los dispositivos removibles automáticamente, puedes usar la siguiente técnica para determinar cómo son nombrados los dispositivos removibles cuando son conectados. Primero, arranca una vista en tiempo real del archivo /var/log/messages o /var/log/syslog (puedes necesitar privilegios de superusuario para ello):

```
[me@linuxbox ~]$ sudo tail -f /var/log/messages
```

Las últimas líneas del archivo se mostrarán y luego se pausarán. A continuación, conecta el dispositivo removible. En este ejemplo, usaremos una memoria flash de 16MB. Casi inmediatamente, el kernel detectará el dispositivo y lo probará:

```
Jul 23 10:07:53 linuxbox kernel: usb 3-2: new full speed USB
device
using uhci_hcd and address 2
Jul 23 10:07:53 linuxbox kernel: usb 3-2: configuration #1 chosen
from 1 choice
Jul 23 10:07:53 linuxbox kernel: scsi3 : SCSI emulation for USB
Mass
Storage devices
Jul 23 10:07:58 linuxbox kernel: scsi scan: INQUIRY result too
short
(5), using 36
Jul 23 10:07:58 linuxbox kernel: scsi 3:0:0:0: Direct-Access Easy
Disk 1.00 PQ: 0 ANSI: 2
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte
hardware sectors (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect
is
off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive
cache: write through
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte
hardware sectors (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect
```

```
is
off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive
cache: write through
Jul 23 10:07:59 linuxbox kernel: sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI
removable disk
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: Attached scsi generic
sg3 type 0
```

Después de que la pantalla se pare de nuevo, pulsa Ctrl-C para volver al prompt. Las partes interesantes de la salida son las referencias repetidas a "[sdb]" que coinciden con nuestra expectativa de un nombre de dispositivo de un disco SCSI. Sabiendo esto, dos líneas pasan a ser particularmente iluminadoras:

```
Jul 23 10:07:59 linuxbox kernel: sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI
removable disk
```

Ésto nos dice que el nombre del dispositivo es `/dev/sdb` para el dispositivo completo y `/dev/sdb1` para la primera partición del dispositivo. Como hemos visto, trabajar con Linux está lleno de ¡Interesante trabajo de detective!

Consejo: usar la técnica `tail -f /var/log/messages` es una gran forma de ver que está haciendo el sistema en el corto plazo.

Con nuestro nombre de dispositivo en la mano, podemos montar la unidad flash:

```
[me@linuxbox ~]$ sudo mkdir /mnt/flash
[me@linuxbox ~]$ sudo mount /dev/sdb1 /mnt/flash
[me@linuxbox ~]$ df
Filesystem 1K-blocks Used     Available Use% Mounted on
/dev/sda2  15115452  5186944  9775164   35%  /
/dev/sda5  59631908  31777376 24776480  57%  /home
/dev/sda1  147764    17277    122858    13%  /boot
tmpfs      776808    0        776808    0%   /dev/shm
/dev/sdb1  15560     0        15560     0%   /mnt/flash
```

El nombre del dispositivo permanecerá igual mientras que permanezca físicamente conectado al ordenador y éste no sea reiniciado.

Creando un nuevo sistema de archivos

Digamos que queremos formatear la unidad flash con un sistema de archivos nativo de Linux, en

lugar del sistema FAT32 que tiene ahora. Ésto implica dos pasos: 1. (opcional) crear una nueva tabla de particiones si la existente no es de nuestro gusto, y 2. crear un sistema de archivos nuevo y vacío en el disco.

¡Advertencia! En el siguiente ejercicio, vamos a formatear una unidad flash. ¡Usa una unidad que no contenga nada que necesites porque será borrada! De nuevo, **asegúrate completamente de que estás especificando el nombre de dispositivo correcto en nuestro sistema, no el que se muestra en el texto. ¡Incumplir esta medida puede provocar el formateo (osea borrado) del disco equivocado!**

Manipulando particiones con fdisk

El programa `fdisk` nos permite interactuar directamente con los dispositivos que se comportan como discos (como discos duros o unidades flash) a muy bajo nivel. Con esta herramienta podemos editar, borrar y crear particiones en el dispositivo. Para trabajar con nuestra unidad flash, tenemos primero que desmontarla (si es necesario) y luego invocar el programa `fdisk` como sigue:

```
[me@linuxbox ~]$ sudo umount /dev/sdb1
[me@linuxbox ~]$ sudo fdisk /dev/sdb
```

Fíjate que tenemos que especificar el dispositivo en términos del dispositivo completo, no por un número de partición. Después de que el programa arranque, veremos el siguiente prompt:

```
Command (m for help):
```

Introducir una “m” mostrará el menú del programa:

```
Command action
a toggle a bootable flag
b edit bsd disklabel
c toggle the dos compatibility flag
d delete a partition
l list known partition types
m print this menu
n add a new partition
o create a new empty DOS partition table
p print the partition table
q quit without saving changes
s create a new empty Sun disklabel
t change a partition's system id
u change display/entry units
v verify the partition table
w write table to disk and exit
x extra functionality (experts only)
```

```
Command (m for help):
```

La primera cosa que queremos hacer es examinar la tabla de particiones existente. Lo hacemos introduciendo “p” para imprimir la tabla de particiones del dispositivo:

```
Command (m for help): p

Disk /dev/sdb: 16 MB, 16006656 bytes
1 heads, 31 sectors/track, 1008 cylinders
Units = cylinders of 31 * 512 = 15872 bytes

Device Boot Start End Blocks Id System
/dev/sdb1 2 1008 15608+ b W95 FAT32
```

En este ejemplo, vemos un dispositivo de 16 MB con una única partición (1) que usa 1006 de los 1008 cilindros disponibles en el dispositivo. La partición está identificada como una partición FAT32 de Windows 95. Algunos programas usarán este identificador para limitar el tipo de operaciones que pueden hacerse en el disco, pero la mayoría de las veces no es crítico cambiarlo. Sin embargo, a modo de demostración, lo cambiaremos para indicar una partición Linux. Para hacerlo, primero debemos encontrar que ID se usa para identificar una partición Linux. En la lista anterior vemos que se usa el ID “b” para especificar la partición existente. Para ver una lista de los tipos de partición disponibles, volveremos al menú del programa. Allí podemos ver la siguiente opción:

```
l list known partition types
```

Si introducimos una “l” en el prompt, se muestra una gran lista de los tipos posibles. Entre ellas vemos “b” para nuestro tipo de partición existente y “83” para Linux.
Volviendo al menú, vemos la opción de cambiar la ID de una partición:

```
t change a partition's system id
```

Introducimos “t” en el prompt y luego la nueva ID:

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 83
Changed system type of partition 1 to 83 (Linux)
```

Esto completa todos los cambios que necesitamos hacer. Hasta ahora, el dispositivo no ha sido tocado (todos los cambios han sido almacenados en la memoria, no en el dispositivo físico), así que escribiremos la tabla de particiones modificada en el dispositivo y saldremos. Para hacerlo,

introducimos “w” en el prompt:

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
[me@linuxbox ~]$
```

Si hubieramos decidido dejar el servicio inalterado, podríamos haber introducido “q” en el prompt, que habría salido del programa sin guardar los cambios. Podemos ignorar tranquilamente el ominoso sonido del mensaje de advertencia.

Creando un nuevo sistema de archivos con mkfs

Con nuestra edición de particiones hecha (más sencillamente de lo que parecía) es hora de crear un nuevo sistema de archivos en nuestra unidad flash. Para hacerlo, usaremos `mkfs` (abreviatura de “make file system” - “fabricar sistema de archivos”), que puede crear sistemas de archivos en una variedad de formatos. Para crear un sistema de archivos ext3 en el dispositivo usamos la opción “-t” para especificar el tipo de sistema “ext3”, seguido por el nombre del dispositivo incluyendo la partición que queremos formatear:

```
[me@linuxbox ~]$ sudo mkfs -t ext3 /dev/sdb1
mke2fs 1.40.2 (12-Jul-2007)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
3904 inodes, 15608 blocks
780 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=15990784
2 block groups
8192 blocks per group, 8192 fragments per group
1952 inodes per group
Superblock backups stored on blocks:
8193

Writing inode tables: done
```

```
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
[me@linuxbox ~]$
```

El programa mostrará un montón de información cuando elegimos el tipo de sistema de archivos ext3. Para reformatear el dispositivos a su sistema de archivos original FAT32, especifica "vfat" como el tipo de sistema de archivos:

```
[me@linuxbox ~]$ sudo mkfs -t vfat /dev/sdb1
```

Este proceso de particionado y formateado puede usarse siempre que añadamos al sistema sistemas de almacenamiento adicionales. Aunque hemos trabajado con una diminuta unidad flash, el mismo proceso puede aplicarse a discos duros internos y otros dispositivos de almacenamiento removibles como discos duros USB.

Probando y reparando sistemas de archivos

En nuestro asunto anterior del archivo `/etc/fstab`, vimos algunos números misteriosos al final de cada línea. Cada vez que el sistema arranca, chequea rutinariamente la integridad del sistema de archivos antes de montarlo. Esto lo hace el programa `fsck` (abreviatura de "file system check" - Chequeo del sistema de archivos). El último número de cada línea de `fstab` especifica el orden en que los dispositivos tienen que ser chequeados. En nuestro ejemplo anterior, vemos que el sistema de archivos raiz es el primero en ser chequeado, seguido del los sistemas de archivos `home` y `boot`. Los dispositivos con un cero cómo último número no son chequeados rutinariamente.

Además del chequeo de la integridad del sistema de archivos, `fsck` puede también reparar sistemas de archivos corruptos con distintos grados de éxito, dependiendo de la magnitud del daño. En sistemas de archivos como-Unix, las porciones recuperadas de archivos se colocan en el directorio `lost+found`, localizado en la raíz de cada sistema de archivos.

Para chequear nuestra unidad flash (que debe ser desmontada primero), podríamos hacer lo siguiente:

```
[me@linuxbox ~]$ sudo fsck /dev/sdb1
fsck 1.40.8 (13-Mar-2008)
e2fsck 1.40.8 (13-Mar-2008)
/dev/sdb1: clean, 11/3904 files, 1661/15608 blocks
```

En mi experiencia, la corrupción del sistema de archivos es muy rara a menos que haya un

problema de hardware, como un disco duro que falle. En la mayoría de los sistemas, la corrupción del sistema archivos detectada en el arranque causará que el sistema se detenga y te lleve a ejecutar `fsck` antes de continuar.

What the fsck?

En la cultura Unix, la palabra "fsck" a menudo se usa en lugar de una popular palabra con la que comparte tres letras. Esto es especialmente apropiado, dado que probablemente pronunciarás la mencionada palabra si te encuentras en una situación en la que estás forzado a ejecutar `fsck`.

Formateando disquetes

Para aquellos de nosotros que todavía usamos ordenadores lo suficientemente antiguos como para que estén equipados con unidades de disquetes, podemos manejar también dichos dispositivos. Preparar un disquete en blanco para usar es un proceso en dos pasos. Primero, realizamos un formateo de bajo nivel en el disquete, y luego creamos un sistema de archivos. Para realizar el formateo, usamos el programa `fdformat` especificando el nombre del dispositivo del disquete (normalmente `/dev/fd0`):

```
[me@linuxbox ~]$ sudo fdformat /dev/fd0
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
```

A continuación, aplicamos el sistema de archivos FAT al disquete con `mkfs`:

```
[me@linuxbox ~]$ sudo mkfs -t msdos /dev/fd0
```

Fíjate que usamos el sistema de archivos de "msdos" para tener las tablas de asignación de archivos más antiguas (y pequeñas). Cuando el disquete esté preparado, se montará como otros dispositivos.

Moviendo datos directamente de/desde dispositivos

Mientras que normalmente pensamos en los datos en nuestros ordenadores organizados en archivos, también es posible pensar en los datos en "bruto". Si miramos un disco duro, por ejemplo, vemos que consiste en un gran números de "bloques" de datos que el sistema operativo ve como directorios y archivos. Sin embargo, si pudiéramos tratar una unidad de disco simplemente como una gran colección de bloques de datos, podríamos realizar tareas útiles, como clonar dispositivos.

El programa `dd` realiza esta tarea. Copia bloques de datos de un lugar a otro. Usa una sintaxis única (por motivos históricos) y se usa normalmente de esta forma:

```
dd if=input_file of=output_file [bs=block_size [count=blocks]]
```

Digamos que tenemos dos unidades USB del mismo tamaño y queremos copiar exactamente la

primera unidad en la segunda. Si conectamos los dos discos al ordenador y están asignados a los dispositivos /dev/sdb y /dev/sdc respectivamente, podríamos copiar todo lo que hay en el primer disco al segundo disco haciendo lo siguiente:

```
dd if=/dev/sdb of=/dev/sdc
```

Alternativamente, si sólo estuviera conectado a nuestro ordenador el primer dispositivo, podríamos copiar su contenido en un archivo normal para restauralo o copiarlo posteriormente:

```
dd if=/dev/sdb of=flash_drive.img
```

¡Aviso! El comando dd es muy potente. Aunque su nombre deriva de "data definition" (definición de datos), a veces se le llama "destroy disk" (destruye discos) porque los usuarios a menudo escriben mal las especificaciones if o of. **¡Siempre comprueba dos veces tus especificaciones de entrada y salida antes de pulsar intro!**

Creando imágenes de CD-ROM

Escribir un CD-ROM grabable (tanto un CD-R como un CD-RW) consta de dos pasos: primero, construir un archivo de imagen iso que es la imagen exacta del sistema de archivos del CD-ROM y segundo, escribir el archivo de imagen en el CD-ROM.

Creando una copia imagen de un CD-ROM

Si queremos hacer una imagen iso de un CD-ROM existente, podemos usar dd para leer todos los bloques de datos del CD-ROM y copiarlos en una archivo local. Digamos que tenemos un CD de Ubuntu y queremos hacer un archivo iso que podamos usar más tarde para hacer más copias. Después de insertar el CD y determinar su nombre de dispositivo (asumiremos que es /dev/cdrom), podemos hacer el archivo iso así:

```
dd if=/dev/cdrom of=ubuntu.iso
```

Esta técnica funciona también para DVDs de datos, pero no funcionará para audio CDs, ya que no usan un sistema de archivos para almacenamiento. Para audio CDs, mira el comando cdrdao.

Creando una imagen de una colección de archivos

Para crear una imagen iso conteniendo el contenido de un directorio, usamos el programa genisoimage. Para hacerlo, primero creamos un directorio conteniendo todos los archivos que queramos incluir en la imagen. Por ejemplo, si hubiéramos creado un directorio llamado ~/cd-rom-files y lo llenamos con archivos para nuestro CD-ROM, podríamos crear un archivo de imagen llamado cd-rom.iso con el siguiente comando:

```
genisoimage -o cd-rom.iso -R -J ~/cd-rom-files
```

La opción “-R” añade metadatos para las *extensiones Rock Ridge*, que permiten usar nombres de archivo largos y permisos de archivos estilo POSIX. Igualmente, la opción “-J” activa las *extensiones Joilet*, que permiten nombres de archivo largos para Windows.

Un programa con otro nombre…

Si miras tutoriales on-line sobre crear y quemar medios ópticos como CD-ROMs y DVDs, frecuentemente encontrarás dos programas llamados `mkisofs` y `cdrecord`. Estos programas eran parte del popular paquete llamado “cdr-tools” creados por Jörg Schilling. En el verano de 2006, Mr. Schilling hizo un cambio de licencia a una parte del paquete cdrtools que, en opinión de muchos en la comunidad Linux, creó una incompatibilidad de licencia con el GNU GPL. Como resultado, surgió un fork del proyecto cdrtools que ahora incluye programas para reemplazar a `cdrecord` y `mkisofs` llamados `wodim` y `genisoimage`, respectivamente.

Escribiendo imágenes de CD-ROM

Una vez que tenemos un archivo de imagen, podemos quemarlo en nuestro medio óptico. La mayoría de los comandos que veamos a continuación pueden ser aplicados tanto a CD-ROM como a DVD.

Montando una imagen ISO directamente

Hay un truco que podemos usar para montar una imagen iso mientras que está todavía en nuestro disco duro y tratarla como si ya estuviera en un medio óptico. Añadiendo la opción “-o loop” a mount (además del requerido tipo de sistema de archivos “-t iso9660”), podemos montar el archivo de imagen como si fuera un dispositivo y conectarlo al árbol del sistema de archivos:

```
mkdir /mnt/iso_image
mount -t iso9660 -o loop image.iso /mnt/iso_image
```

En el ejemplo anterior, creamos un punto de montaje llamado `/mnt/iso_image` y luego montamos el archivo de imagen `image.iso` en el punto de montaje. Una vez que la imagen está montada, puede ser tratada como si fuera un CD-ROM o DVD real. *Recuerda desmontar la imagen cuando ya no la necesites.*

Borrando un CD-ROM regrabable

Los CD-RW regrabables necesitan ser *borrados* antes de ser reutilizados. Para hacerlo, podemos usar `wodim`, especificando el nombre del dispositivo del grabador de CD y el tipo de borrado a realizar.

El programa `wodim` ofrece varios tipos. El más simple (y rápido) es el modo “rápido”:

```
wodim dev=/dev/cdrw blank=fast
```

Escribiendo una imagen

Para escribir una imagen, usamos de nuevo `wodim`, especificando el nombre del dispositivo grabador óptico y el nombre del archivo de imagen:

```
wodim dev=/dev/cdrw image.iso
```

Además del nombre del dispositivo y el archivo de imagen, `wodim` soporta una gran variedad de opciones. Dos muy comunes son "-v" para salida verbose, y "-dao", que graba el disco en el modo *disc-at-once* (disco de una vez). Este modo debería usarse si estás preparando un disco para reproducción comercial. El modo por defecto de `wodim` es *track-at-one* (pista de una vez), que es útil para grabar pistas de música.

Resumiendo

En este capítulo hemos visto las tareas básicas de la gestión del almacenamiento. Hay, por supuesto, muchas más. Linux soporta una gran cantidad de dispositivos de almacenamiento y tablas de sistemas de archivos. También ofrece funciones para interoperar con otros sistemas.

Para saber más

Hecha un vistazo a las man pages de los comandos que hemos visto. Algunos de ellos soportan gran número de opciones y operaciones. También, busca tutoriales on-line para añadir discos duros a tu sistema Linux (hay muchos) y funcionan con medios ópticos.

Crédito adicional

A menudo es útil verificar la integridad de una imagen iso que hemos descargado. En la mayoría de los casos, un distribuidor de una imagen iso también proporcionará un archivo checksum. Un checksum es el resultado de un calculo matemático exótico que tiene como resultado un número que representa el contenido del archivo en cuestión. Si el contenido del archivo cambia aunque sea en un sólo bit, el resultado del checksum será muy diferente. El método más común de generación de checksum usa el programa `md5sum`. Cuando usas `md5sum`, produce un número hexadecimal único:

```
md5sum image.iso
34e354760f9bb7fbf85c96f6a3f94ece image.iso
```

Después de descargar la imagen, deberías ejecutar `md5sum` contra ella y comparar el resultado con el valor de `md5sum` suministrado por el publicador.

Además de chequear la integridad de un archivo descargado, podemos usar md5sum para verificar el nuevo medio óptico grabado. Para hacerlo, primero calculamos el checksum del archivo de imagen y luego calculamos el checksum del medio óptico. El truco para verificar el medio óptico es limitar el cálculo a sólo una la parte del medio que contiene la imagen. Hacemos ésto determinando el número de bloques de 2048 bytes que contiene la imagen (los medios ópticos están siempre

escritos en bloques de 2048 bytes) y leyendo todos esos bloques del medio. En algunos tipos de medios, ésto no es necesario. Un CD-R grabado en modo disc-at-once puede ser chequeado de esta forma:

```
md5sum /dev/cdrom
34e354760f9bb7fbf85c96f6a3f94ece /dev/cdrom
```

Muchos tipos de medios, como DVDs, requieren un cálculo preciso del número de bloques. En el ejemplo siguiente, chequeamos la integridad del archivo de imagen `dvd-image.iso` y del disco en el lector de DVD `/dev/dvd`. ¿Puedes adivinar como funciona?

```
md5sum dvd-image.iso; dd if=/dev/dvd bs=2048 count=$(( $(stat -c
"%s"

dvd-image.iso) / 2048 )) | md5sum
```

Redes

Cuando llegamos a las redes, probablemente no hay nada que no pueda hacerse con Linux. Linux se usa para construir todo tipo de sistemas y aplicaciones de red, incluyendo cortafuegos, routers, servidores de nombres, NAS (Network Attached Storage - Almacenamiento conectado en red) y más y más.

Como el tema de la redes es amplio, también lo es el número de comandos que pueden usarse para configurarlas y controlarlas. Centraremos nuestra atención en sólo unos pocos de los más frecuentemente usados. Los comandos que hemos elegido examinar son los usados para monitorizar redes y los usados para transferir archivos. Además, vamos a explorar el programa `ssh` que se usa para realizar accesos remotos. Este capítulo cubrirá:

- `ping` – Envía un ICMP ECHO_REQUEST a hosts de la red
- `traceroute` – Imprime la traza de la ruta de los paquetes hasta un host de la red
- `netstat` – Imprime conexiones de red, tablas de rutas, estadísticas de interfaz, conexiones enmascaradas y pertenencias a multicasts
- `ftp` – Programa de transferencia de archivos de Internet
- `wget` – Descargas de red no interactivas
- `ssh` – Cliente SSH OpenSSH (programa de acceso remoto)

Vamos a asumir que tenemos algo de experiencia en redes. En esta era de Internet, todo el que use un ordenador necesita un entendimiento básico de los conceptos de redes. Para hacer un uso completo de este capítulo deberíamos familiarizarnos con los siguientes términos:

- Dirección IP (Internet Protocol)
- Nombres de host y de dominio
- URI (Uniform Resource Identifier – Identificador de recursos uniforme)

Por favor mira la siguiente sección “Para saber más” para ver algunos artículos útiles respecto a estos términos.

Nota: Algunos de los comandos que veremos pueden (dependiendo de tu distribución) requerir la instalación de paquetes adicionales de los repositorios de tu distribución, y algunos pueden requerir

privilegios de superusuario para ejecutarlos.

Examinando y monitorizando una red

Incluso si no eres el administrador del sistema, a menudo es útil examinar el rendimiento y funcionamiento de una red.

ping

El comando de red más básico es `ping`. El comando `ping` envía un paquete de red especial llamado IMCP ECHO_REQUEST a un host especificado. La mayoría de los dispositivos de red que reciban estos paquetes responderán, permitiendo verificar la conexión de red.

Nota: es posible configurar la mayoría de los dispositivos de red (incluyendo los host Linux) para ignorar estos paquetes. Esto se hace normalmente por razones de seguridad, para ocultar parcialmente un host de un atacante potencial. También es común configurar los cortafuegos para bloquear el tráfico IMCP.

Por ejemplo, para ver si podemos alcanzar `linuxcommand.org` (uno de nuestros sitios favoritos ;-), podemos usar `ping` de la siguiente forma:

```
[me@linuxbox ~]$ ping linuxcommand.org
```

Una vez que empieza, `ping` continua enviando paquetes con un intervalo específico (por defecto es un segundo) hasta que lo interrumpamos:

```
[me@linuxbox ~]$ ping linuxcommand.org
PING linuxcommand.org (66.35.250.210) 56(84) bytes of data.
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=1
ttl=43 time=107 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=2
ttl=43 time=108 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=3
ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=4
ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=5
ttl=43 time=105 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=6
ttl=43 time=107 ms
--- linuxcommand.org ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 6010ms
rtt min/avg/max/mdev = 105.647/107.052/108.118/0.824 ms
```

Después de interrumpirlo (en este caso después del sexto paquete) pulsando `Ctrl-c`, `ping` muestra las estadísticas de funcionamiento. Un funcionamiento apropiado de la red mostrará cero por ciento de paquetes perdidos. Un “ping” exitoso indicará que los elementos de la red (sus tarjetas de red, calbes, routas y puertas de enlaces) están, en general, funcionando correctamente.

traceroute

El programa `traceroute` (algunos sistemas usan el similar `tracepath` en su lugar) muestra

una lista de todos los “hops” o saltos que el tráfico de red hace para llegar desde el sistema local al host especificado. Por ejemplo, para ver la ruta tomada para alcanzar `slashdot.org`, haríamos esto:

```
[me@linuxbox ~]$ traceroute slashdot.org
```

La salida tiene esta pinta:

```
traceroute to slashdot.org (216.34.181.45), 30 hops max, 40 byte
packets
1 ipcop.localdomain (192.168.1.1) 1.066 ms 1.366 ms 1.720 ms
2 * * *
3 ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9) 14.622
ms 14.885 ms 15.169 ms
4 po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154) 17.634
ms 17.626 ms 17.899 ms
5 po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158) 15.992
ms 15.983 ms 16.256 ms
6 po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5) 22.835
ms 14.233 ms 14.405 ms
7 po-10-ar02.whitemarsh.md.bad.comcast.net (68.87.129.34) 16.154
ms 13.600 ms 18.867 ms
8 te-0-3-0-1-cr01.philadelphia.pa.ibone.comcast.net (68.86.90.77)
21.951 ms 21.073 ms 21.557 ms
9 pos-0-8-0-0-cr01.newyork.ny.ibone.comcast.net (68.86.85.10)
22.917 ms 21.884 ms 22.126 ms
10 204.70.144.1 (204.70.144.1) 43.110 ms 21.248 ms 21.264 ms
11 cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93) 21.857 ms
cr2-pos-0-0-3-1.newyork.savvis.net (204.70.204.238) 19.556 ms cr1-
pos-0-7-3-1.newyork.savvis.net (204.70.195.93) 19.634 ms
12 cr2-pos-0-7-3-0.chicago.savvis.net (204.70.192.109) 41.586 ms
42.843 ms cr2-tengig-0-0-2-0.chicago.savvis.net (204.70.196.242)
43.115 ms
13 hr2-tengigabitethernet-12-1.elkgrovech3.savvis.net
(204.70.195.122) 44.215 ms 41.833 ms 45.658 ms
14 csr1-ve241.elkgrovech3.savvis.net (216.64.194.42) 46.840 ms
43.372 ms 47.041 ms
15 64.27.160.194 (64.27.160.194) 56.137 ms 55.887 ms 52.810 ms
16 slashdot.org (216.34.181.45) 42.727 ms 42.016 ms 41.437 ms
```

En la salida, podemos ver que conectar desde nuestro sistema a `slashdot.org` requiere atravesar dieciséis routers. Cuatro routers que dan información identificativa, vemos sus nombres, direcciones IP, y datos de funcionamiento, que incluyen tres ejemplos de viajes de ida y vuelta desde el sistema local a router. Para los routers que no ofrecen información identificativa (debido a la configuración del router, congestión en la red, cortafuegos, etc.) vemos asteriscos como en la línea del salto número 2.

netstat

El programa `netstat` se usa para examinar varias configuraciones de red y estadísticas. Mediante el uso de sus muchas opciones, podemos ver una variedad de características de nuestra configuración de red. Usando la opción “-ie”, podemos examinar las interfaces de red de nuestro

sistema:

```
[me@linuxbox ~]$ netstat -ie
eth0 Link encap:Ethernet HWaddr 00:1d:09:9b:99:67
inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:238488 errors:0 dropped:0 overruns:0 frame:0
TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:100
RX bytes:153098921 (146.0 MB) TX bytes:261035246 (248.9 MB)
Memory:fdfc0000-fdfe0000
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:16436 Metric:1
RX packets:2208 errors:0 dropped:0 overruns:0 frame:0
TX packets:2208 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:111490 (108.8 KB) TX bytes:111490 (108.8 KB)
```

En el ejemplo anterior, vemos que nuestro sistema de prueba tiene dos interfaces de red. La primera, llamada `eth0`, es la interfaz Ethernet, y la segunda, llamada `lo`, es la *interfaz loopback*, una interfaz virtual que el sistema utiliza para “hablar consigo mismo”.

Cuando realizamos un diagnostico ocasional de la red, las cosas más importantes a revisar son la presencia de la palabra “UP” al principio de la cuarta linea de cada interfaz, indicando que la interfaz de red está disponible, y la presencia de una IP válida en el campo `inet addr` de la segunda línea. Para sistemas que usen DHCP (Dynamic Host Configuration Protocol – Protocolo de configuración dinámica de host), una IP válida en este campo verificará que el DHCP está funcionando.

Usando la opción “-r” mostrará la tabla de rutas de red del kernel. Ésto muestra como está configurada la red para enviar paquetes de red a red:

```
[me@linuxbox ~]$ netstat -r
Kernel IP routing table
Destination  Gateway      Genmask       Flags MSS Window irtt
Iface
192.168.1.0  *            255.255.255.0 U     0   0      0    eth0
default      192.168.1.1  0.0.0.0       UG    0   0      0    eth0
```

En este ejemplo sencillo, vemos un tabla de rutas típica de una máquina cliente en una LAN (Local Area Network – Red de Área Local) detrás de un cortafuegos/router. La primera línea de la lista muestra el destino `192.168.1.0`. Las direcciones IP terminadas en cero se refieren a redes en lugar de a hosts individuales, así que este destino significa cualquier host de la LAN. El siguiente campo, `Gateway`, es el nombre de la dirección IP de la puerta de enlace (router) usada para ir del host actual a la red de destino. Un asterisco en este campo indica que no se necesita puerta de enlace.

La última línea contiene el destino `default`. Ésto significa todo el tráfico destinado a una red que no esté listada de ninguna forma en la tabla. En nuestro ejemplo, vemos que la puerta de enlace está

definida como un router con la dirección `192.168.1.1`, que presumiblemente sabe que hacer con el destino del tráfico.

El programa `netstat` tiene muchas opciones y sólo hemos visto un par. Échale un vistazo a la man page de `netstat` para una lista completa.

netstat

El programa `netstat` se usa para examinar varias configuraciones de red y estadísticas. Mediante el uso de sus muchas opciones, podemos ver una variedad de características de nuestra configuración de red. Usando la opción "-ie", podemos examinar las interfaces de red de nuestro sistema:

```
[me@linuxbox ~]$ netstat -ie
eth0 Link encap:Ethernet HWaddr 00:1d:09:9b:99:67
inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:238488 errors:0 dropped:0 overruns:0 frame:0
TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:100
RX bytes:153098921 (146.0 MB) TX bytes:261035246 (248.9 MB)
Memory:fdfc0000-fdfe0000
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:16436 Metric:1
RX packets:2208 errors:0 dropped:0 overruns:0 frame:0
TX packets:2208 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:111490 (108.8 KB) TX bytes:111490 (108.8 KB)
```

En el ejemplo anterior, vemos que nuestro sistema de prueba tiene dos interfaces de red. La primera, llamada `eth0`, es la interfaz Ethernet, y la segunda, llamada `lo`, es la *interfaz loopback*, una interfaz virtual que el sistema utiliza para "hablar consigo mismo".

Cuando realizamos un diagnostico ocasional de la red, las cosas más importantes a revisar son la presencia de la palabra "UP" al principio de la cuarta linea de cada interfaz, indicando que la interfaz de red está disponible, y la presencia de una IP válida en el campo `inet addr` de la segunda línea. Para sistemas que usen DHCP (Dynamic Host Configuration Protocol – Protocolo de configuración dinámica de host), una IP válida en este campo verificará que el DHCP está funcionando.

Usando la opción "-r" mostrará la tabla de rutas de red del kernel. Ésto muestra como está configurada la red para enviar paquetes de red a red:

```
[me@linuxbox ~]$ netstat -r
Kernel IP routing table
Destination  Gateway      Genmask        Flags MSS Window irtt
Iface
192.168.1.0  *            255.255.255.0 U      0   0       0    eth0
```

```
default      192.168.1.1  0.0.0.0       UG    0   0      0    eth0
```

En este ejemplo sencillo, vemos un tabla de rutas típica de una máquina cliente en una LAN (Local Area Network – Red de Área Local) detrás de un cortafuegos/router. La primera línea de la lista muestra el destino `192.168.1.0`. Las direcciones IP terminadas en cero se refieren a redes en lugar de a hosts individuales, así que este destino significa cualquier host de la LAN. El siguiente campo, `Gateway`, es el nombre de la dirección IP de la puerta de enlace (router) usada para ir del host actual a la red de destino. Un asterisco en este campo indica que no se necesita puerta de enlace.

La última línea contiene el destino `default`. Ésto significa todo el tráfico destinado a una red que no esté listada de ninguna forma en la tabla. En nuestro ejemplo, vemos que la puerta de enlace está definida como un router con la dirección `192.168.1.1`, que presumiblemente sabe que hacer con el destino del tráfico.

El programa `netstat` tiene muchas opciones y sólo hemos visto un par. Échale un vistazo a la man page de `netstat` para una lista completa.

ftp

Uno de los auténticos programas "clásicos", `ftp` tomo su nombre del protocolo que usa, el *File Transfer Protocol (Protocolo de transferencia de ficheros)*. Ftp se usa ampliamente en Internet para descarga de archivos. La mayoría, si no todos, los navegadores web lo soportan y a menudo verás URIs que comienzan por el protocolo `ftp://`

Antes de que hubiera navegadores web, había programas `ftp`. `ftp` se usa para comunicar *servidores FTP*, máquinas que contienen archivos que pueden ser subidos y descargados a través de una red.
FTP (en su forma original) no es seguro, porque envía nombres de cuenta y contraseñas en *texto plano*. Ésto significa que no están encriptados y que cualquiera que esnife la red puede verlos. Debido a ésto, casi todos los FTP a través de Internet están ofrecidos por *servidores anónimos de FTP*. Un servidor anónimo permite que cualquiera acceda usando el nombre de usuario "annonymous" y una contraseña sin sentido.

En el ejemplo siguiente, mostramos una sesión típica con el programa `ftp` descargando una imagen iso Ubuntu localizada en el directorio `/pub/cd_images/Ubuntu-8.04` del servidor FTP anónimo `fileserver`:

```
[me@linuxbox ~]$ ftp fileserver
Connected to fileserver.localdomain.
220 (vsFTPd 2.0.1)
Name (fileserver:me): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/cd_images/Ubuntu-8.04
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
```

```
150 Here comes the directory listing.
-rw-rw-r-- 1 500 500 733079552 Apr 25 03:53 ubuntu-8.04-
desktop-i386.iso
226 Directory send OK.
ftp> lcd Desktop
Local directory now /home/me/Desktop
ftp> get ubuntu-8.04-desktop-i386.iso
local: ubuntu-8.04-desktop-i386.iso remote: ubuntu-8.04-
desktopi386.
iso
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for ubuntu-8.04-
desktopi386.
iso (733079552 bytes).
226 File send OK.
733079552 bytes received in 68.56 secs (10441.5 kB/s)
ftp> bye
```

Aquí hay una explicación de los comandos introducidos durante esta sesión:

Comando	**Significado**
`ftp fileserver`	Invoca el programa `ftp` y se conecta al servidor FTP `fileserver`.
`annonymous`	Nombre de acceso. Después del prompt de login, aparecerá un prompt de contraseña. Algunos servidores aceptarán una contraseña en blanco, otros requerirán una contraseña en forma de dirección de correo electrónico. En este caso, prueba algo como “user@example.com”.
`cd pub/cd_images/Ubuntu-8.04`	Cambia al directorio en el sistema remoto que contiene el archivo que queremos. Fíjate que en la mayoría de los servidores de FTP anónimos, los archivos para descarga pública se encuentran en algún lugar del directorio `pub`.
`ls`	Lista el directorio del sistema remoto.
`lcd Desktop`	Cambia el directorio en el sistema local a `~/Desktop`. En el ejemplo, el programa ftp fue invocado cuando el directorio de trabajo era `~`. Este comando cambia el directorio de trabajo a `~/Desktop`.
`get ubuntu-8.04-desktopi386. iso`	Le dice al sistema remoto que transfiera el archivo `ubuntu-8.04-desktopi386.iso` al sistema local. Como el directorio de trabajo del sistema local se cambió a `~/Desktop`, el archivo será descargado allí.

bye	Se desconecta del sistema remoto y termina la sesión del programa `ftp`. Los comandos `quit` y `exit` también pueden utilizarse.

Escribir "help" en el prompt "ftp>" mostrará una lista de los comandos soportados. Usando ftp en un servidor donde tengamos suficientes permisos, es posible realizar muchas tareas ordinarias de gestión de archivos. Es tosco, pero funciona.

lftp – Un ftp mejor

`ftp` no es el único cliente FTP en línea de comandos. De hecho, hay muchos. Uno de los mejores (y más populares) es `lftp` de Alexander Lukyanov. Funciona de forma muy parecida al programa `ftp` tradicional, pero tiene muchas características convenientes adicionales incluyendo soporte de protocolo múltiple (incluyendo HTTP), reintento automático de descargas fallidas, procesos en segundo plano, completado con el tabulador de las rutas, y mucho más.

wget

Otro programa popular de la línea de comandos para descargar archivos es `wget`. Es útil para descargar contenido tanto de sitios web como ftp. Archivos independientes, archivos múltiples e incluso sitios enteros puede ser descargados. Para descargar la primera página de `linuxcommand.org` podríamos hacer esto:

```
[me@linuxbox ~]$ wget http://linuxcommand.org/index.php
--11:02:51-- http://linuxcommand.org/index.php
=> `index.php'
Resolving linuxcommand.org... 66.35.250.210
Connecting to linuxcommand.org|66.35.250.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
[ <=> ] 3,120 --.--K/s
11:02:51 (161.75 MB/s) - `index.php' saved [3120]
```

Las muchas opciones del programa permite a `wget` la descarga recursiva, descargar archivos en segundo plano (permitiéndote desconectarte pero seguir descargando), y completar la descarga de un archivo parcialmente descargado. Estas características están bien documentadas en su man page que es superior a la media.

Comunicación segura con hosts remotos

Por muchos años, los sistemas operativos como-Unix han tenido la habilidad de ser administrados remotamente a través de una red. En los primeros tiempos, antes de la adopción general de Internet, había un par de programas populares usados para conectarse a hosts remotos. Eran los programas `rlogin` y `telnet`. Estos programas, sin embargo, tienen el mismo fallo que el programa `ftp`; transmiten todas sus comunicaciones (incluyendo nombres de usuario y contraseñas) en texto plano. Esto los hace totalmente inapropiados para usarlos en la era de Internet.

ssh

Para encaminar este problema, un nuevo protocolo llamado SSH (Secure Shell) fue desarrollado.

SSH soluciona los dos problemas básicos de la comunicación segura con un host remoto. Primero, comprueba que el host remoto es quien dice ser (previniendo los llamados ataques de "hombre en el medio"), y segundo, encripta todas las comunicaciones entre los host local y remoto.

SSH consta de dos partes. Un servidor SSH funciona en el host remoto, escuchando conexiones entrantes en el puerto 22, mientras que un cliente SSH se usa en el sistema local para comunicarse con el servidor remoto.

La mayoría de las distribuciones Linux están equipadas con una implementación de SSH llamada OpenSSH del proyecto OpenBSD. Algunas distribuciones incluyen los paquetes tanto del cliente como del servidor por defecto (por ejemplo, Red Hat), mientras que otras (como Ubuntu) solo tiene el cliente. Para permitir que un sistema reciba conexiones remotas, debe tener instalado el paquete `OpenSSH-server`, configurado y ejecutándose, y (si el sistema está también ejecutando un cortafuegos o tras él) debe permitir las conexiones de red entrantes por el puerto TCP 22.

Consejo: Si no tienes un servidor remoto para conectarte pero quieres probar estos ejemplos, asegurate de que el paquete `OpenSSH-server` está instalado en tu sistema y usa `localhost` como nombre del host remoto. De esta forma, tu máquina creará conexiones de red consigo misma.

El cliente SSH usado para conectar con servidores SSH remotos se llama, muy apropiadamente, ssh. Para conectar a un host remoto llamado `remote-sys`, usaríamos el cliente ssh así:

```
[me@linuxbox ~]$ ssh remote-sys
The authenticity of host 'remote-sys (192.168.1.4)' can't be
established.
RSA key fingerprint is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Are you sure you want to continue connecting (yes/no)?
```

La primera vez que se intenta la conexión, se muestra un mensaje indicando que la autenticidad del host remoto no puede ser establecida. Es porque el programa cliente nunca ha visto al host remoto antes. Para aceptar las credenciales del host remoto, introduce "yes" cuando te pregunte. Una vez que se establece la conexión, el usuario/a es preguntado/a por su contraseña:

```
Warning: Permanently added 'remote-sys,192.168.1.4' (RSA) to the
list
of known hosts.
me@remote-sys's password:
```

Después de que introduzcamos la contraseña correctamente, recibimos el prompt de shell del sistema remoto:

```
Last login: Sat Aug 30 13:00:48 2008
[me@remote-sys ~]$
```

La sesión remota de shell continua hasta que el usuario introduzca el comando `exit` en el prompt de shell remoto, cerrando de ese modo la conexión remota. En este punto, la sesión local de shell vuelve y reaparece el prompt de shell local.

También es posible conectar a sistemas remotos usando un nombre de usuario diferente. Por ejemplo, si el usuario local "me" tiene una cuenta llamada "bob" en un sistema remoto, el usuario me podría acceder a la cuenta bob en el sistema remoto de la siguiente forma:

```
[me@linuxbox ~]$ ssh bob@remote-sys
bob@remote-sys's password:
Last login: Sat Aug 30 13:03:21 2008
[bob@remote-sys ~]$
```

Como vimos antes, ssh verifica la autenticidad del host remoto. Si el host remoto no se autentica correctamente, aparece el siguiente mensaje:

```
[me@linuxbox ~]$ ssh remote-sys
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@     WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!    @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle
attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Please contact your system administrator.
Add correct host key in /home/me/.ssh/known_hosts to get rid of
this
message.
Offending key in /home/me/.ssh/known_hosts:1
RSA host key for remote-sys has changed and you have requested
strict
checking.
Host key verification failed.
```

Este mensaje está causado por una de dos posibles situaciones. Primero, un atacante habría intentado un ataque “hombre en el medio”. Es raro, ya que todo el mundo sabe que ssh alerta al usuario de ello. El culpable más probable es que el sistema remoto haya cambiado de alguna forma, por ejemplo, su sistema operativo o su servidor SSH ha sido reinstalado. En interés de la seguridad sin embargo, la primera posibilidad no debería descartarse. Siempre comprueba con el administrador del sistema remoto cuando aparezca este mensaje.

Tras determinar que el mensaje se debe a una causa benigna, es más seguro corregir el problema en el lado del cliente. Ésto se hace usando el editor de texto (quizás vim) para eliminar las claves obsoletas del archivo `~/.ssh/know_host.` En el siguiente mensaje de ejemplo, vemos esto:

```
Offending key in /home/me/.ssh/known_hosts:1
```

Ésto significa que la línea uno del archivo known_host contiene la clave infractora. Borra esta línea del archivo y el programa ssh podrá aceptar nuevas credenciales del sistema remoto.

Además de abrir una sesión de shell en el sistema remoto, ssh también nos permite ejecutar un único comando en un sistema remoto. Por ejemplo, para ejecutar el comando free en un sistema remoto llamado remote-sys y mostrar el resultado en el sistema local:

```
[me@linuxbox ~]$ ssh remote-sys free
me@twin4's password:
total used free shared buffers cached
Mem: 775536 507184 268352 0 110068 154596
-/+ buffers/cache: 242520 533016
Swap: 1572856 0 1572856
[me@linuxbox ~]$
```

Es posible usar esta técnica de muchas formas interesantes, como en este ejemplo en que realizamos un ls en el sistema remoto y redirigimos la salida a un archivos en el sistema local:

```
[me@linuxbox ~]$ ssh remote-sys 'ls *' > dirlist.txt
me@twin4's password:
[me@linuxbox ~]$
```

Fíjate que se usan comillas simples en el comando anterior. Ésto es porque no queremos que se produzca la expansión del nombre en la máquina local; en su lugar, queremos que se realice en el sistema remoto. Igualmente, si hubiéramos querido redirigir la salida a un archivo de la máquina remota, podríamos haber colocado el operador de redirección y el nombre de archivo dentro de las comillas simples:

```
[me@linuxbox ~]$ ssh remote-sys 'ls * > dirlist.txt'
```

Tunelización con SSH

Parte de lo que ocurre cuando estableces una conexión con un host remoto via SSH es que un *túnel encriptado* se crea entre los sistemas local y remoto. Normalmente, este túnel se usa para permitir que los comandos escritos en el sistema local sean transmitidos de forma segura al sistema remoto, y que los resultados sean transmitidos de forma segura de vuelta. Además de esta función básica, el protocolo SSH permite que la mayoría de los tipos de tráfico de red sean enviados a través de un túnel encriptado, creando una especie de *VPN* (Virtual Private Network – Red Privada Virtual) entre los sistemas local y remoto.

Quizás el uso más común de esta característica es permitir que el tráfico de un sistema X Window sea transmitido. En un sistema corriendo un servidor X (o sea, una máquina que muestra una GUI), es posible arrancar y ejecutar un programa cliente X (una aplicación gráfica) en un sistema remoto y hacer que su pantalla aparezca en el sistema local. Es fácil de hacer; aquí tenemos un ejemplo: Digamos que estamos sentados ante un sistema Linux llamado `linuxbox` que está ejecutando un servidor X, y queremos ejecutar el programa xload un un sistema remoto llamado `remote-sys` y ver la salida gráfica del programa en nuestro sistema local. Podríamos hacer ésto:

```
[me@linuxbox ~]$ ssh -X remote-sys
me@remote-sys's password:
Last login: Mon Sep 08 13:23:11 2008
[me@remote-sys ~]$ xload
```

Después de que se ejecute el comando `xload` en el sistema remoto, su ventana

aparece en el sistema local. En algunos sistemas, necesitarías usar la opción "-Y" en lugar de la opción "-X" para hacerlo.

cp y sftp

El paquete OpenSSH también incluye dos programas que pueden hacer uso de un tunel encriptado SSH para copiar archivos a través de una red. El primero, `scp` (secure copy – copia segura) se usa de forma muy parecida al familiar programa `cp` para copiar archivos. La diferencia más notable es que la ruta de origen y destino tienen que ir precedidas por el nombre de un host remoto, seguido de un punto. Por ejemplo, si quisiéramos copiar un documento llamado `document.txt` de nuestro directorio home del sistema remoto, `remote-sys`, al directorio de trabajo actual de nuestro sistema local, podríamos hacer ésto:

```
[me@linuxbox ~]$ scp remote-sys:document.txt .
me@remote-sys's password:
document.txt 100% 5581 5.5KB/s 00:00
[me@linuxbox ~]$
```

Como con ssh, tienes que aplicar un nombre de usuario al prinicipio del nombre del host remoto si el nombre de usuario del host remoto deseado no coincide con el del sistema local:

```
[me@linuxbox ~]$ scp bob@remote-sys:document.txt .
```

El segundo programa para copiado de archivos SSH es `sftp`, el cual, como su nombre indica, es un sustituto seguro del programa `ftp`. `sftp` funciona muy parecido al programa `ftp` original que usamos antes; sin embargo, en lugar de transmitir todo en texto plano, usa un túnel SSH ecriptado. `sftp` tiene una ventaja importante sobre el `ftp` convencional ya que no requiere un servidor FTP corriendo en el host remoto. Sólo requiere el servidor SSH. Ésto significa que cualquier máquina remota que pueda conectar con el cliente SSH puede también ser usada como un servidor como-FTP. Aquí hay una sesión de ejemplo:

```
[me@linuxbox ~]$ sftp remote-sys
Connecting to remote-sys...
me@remote-sys's password:
sftp> ls
ubuntu-8.04-desktop-i386.iso
sftp> lcd Desktop
sftp> get ubuntu-8.04-desktop-i386.iso
Fetching /home/me/ubuntu-8.04-desktop-i386.iso to ubuntu-8.04-
desktop-i386.iso
/home/me/ubuntu-8.04-desktop-i386.iso 100% 699MB 7.4MB/s 01:35
sftp> bye
```

Consejo: El protocolo SFTP es soportado por la mayoría de los gestores gráficos de archivos que encontramos en las distribuciones Linux. Tanto si usamos Nautilus (GNOME) como Konqueror (KDE), podemos introducir una URI comenzando por sftp:// en la barra de direcciones y operar con archivos almacenados en un sistema remoto que esté ejecutando un servidor SSH.

¿Un cliente SSH para Windows?

Digamos que estás sentado en una máquina Windows pero necesitas acceder a tu servidor Linux para hacer algún trabajo; ¿qué puedes hacer? ¡Conseguir un programa cliente SSH para tu

Windows, claro! Hay varios. El más popular es probablemente PuTTY de Simon Tatham y su equipo. El programa PuTTY muestra una ventana de terminal y permite que un usuario Windows abra una sesión SSH (o telnet) en un host remoto. El programa también cuenta con funciones análogas para los programas scp y sftp.

PuTTY está disponible en http://www.chiark.greenend.org.uk/~sgtatham/putty/

Resumiendo

En este capítulo, hemos inspeccionado el campo de las herremientas de red de la mayoría de los sistemas Linux. Como Linux se usa tan extensamente en servidores y aplicaciones de red, hay muchos más que pueden añadirse instalando software adicional. Pero incluso con la colección basica de herramientas, es posible realizar muchas tareas útiles relacionadas con las redes.

Para saber más

- Para una amplia (aunque anticuada) vista de la administración de redes, el Linux Documentation Project (Proyecto de Documentación de Linux) ofrece la Linux Network Administrator's Guide (Guía del Administrador de Redes Linux): http://tldp.org/LDP/nag2/index.html
- Wikipedia contiene muchos buenos artículos sobre redes. Aquí tienes algunos de los más básicos:
  http://en.wikipedia.org/wiki/Internet_protocol_address
  http://en.wikipedia.org/wiki/Host_name
  http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

Buscando archivos

A medida que hemos ido deambulando por nuestro sistema Linux, una cosa ha quedado extremadamente clara: Un sistema Linux típico ¡Tiene un montón de archivos! Esto provoca la pregunta, "¿Cómo podemos encontrar cosas?" Ya sabemos que el sistema de archivos de Linux está bien organizado según convenios que han pasado de generación en generación de los sistemas como-Unix, pero el gran número de archivos puede representar un problema desalentador.

En este capítulo, veremos dos herramientas que se utilizan para encontrar archivos en un sistema. Estas herramientas son:

- `locate` – Encuentra archivos por el nombre
- `find` – Busca archivos en una jerarquía de directorios

También veremos un comando que a menudo se usa con comandos de búsqueda de archivos para procesar la lista resultante de archivos:

- `xargs` – Construye y ejecuta líneas de comando desde la entrada estándar

Además, presentaremos un par de comandos para ayudarnos en nuestras exploraciones:

- `touch` – Cambia la hora de los archivos
- `stat` – Muestra el estado de archivos y sistemas de archivos

locate – Buscar archivos de forma fácil

El programa `locate` realiza una búsqueda rápida en la base de datos de nombres de archivo, y luego muestra cada nombre que coincida con una cadena dada. Digamos, por ejemplo, que

queremos buscar todos los programas con nombres que empiecen por “zip”. Como estamos buscando programas, podemos asumir que el nombre del directorio que contiene los programas terminará con “bin/”. Por lo tanto, podríamos intentar usar `locate` de esta forma para buscar nuestros archivos:

```
[me@linuxbox ~]$ locate bin/zip
```

`locate` buscará en la base de datos de nombres de archivo y mostrará todo aquel que contenga la cadena “bin/zip”.

```
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

Si los requisitos de la búsqueda no son tan simples, `locate` puede combinarse con otras herramientas como `grep` para diseñar búsquedas más interesantes:

```
[me@linuxbox ~]$ locate zip | grep bin
/bin/bunzip2
/bin/bzip2
/bin/bzip2recover
/bin/gunzip
/bin/gzip
/usr/bin/funzip
/usr/bin/gpg-zip
/usr/bin/preunzip
/usr/bin/prezip
/usr/bin/prezip-bin
/usr/bin/unzip
/usr/bin/unzipsfx
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

El programa `locate` ha estado por aquí desde hace muchos años, y hay varias variantes diferentes de uso común. Las dos más comunes que encontramos en las distribuciones de Linux modernas son `slocate` y `mlocate`, aunque a menudo son accesibles por un enlace simbólico llamado `locate`. Las diferentes versiones de `locate` tienen opciones superpuestas. Algunas versiones incluyen coincidencia de expresiones regulares (que veremos en un capítulo próximo) y soporte de comodines. Mira la man page de locate para determinar que versión de `locate` tienes instalada.

¿De donde sale la base de datos de locate?

Habrás notado que, en algunas distribuciones, `locate` falla al funcionar justo después de instalar el sistema, pero si lo intentas de nuevo al día siguiente, funciona bien. ¿Qué pasa? La base de datos

de locate la crea otro programa llamado updatedb. Normalmente, se ejecuta periódicamente como un *trabajo programado*; o sea, una tarea realizada a intervalos regulares por demonio cron. La mayoría de los sistemas equipados con locate ejecutan updatedb una vez al día. Como la base de datos no se actualiza continuamente, notarás que los archivos muy recientes no aparecen cuando usas locate. Para solucionar esto, es posible ejecutar el programa updatedb manualmente entrando como superusuario y ejecutando updatedb en el prompt.

find – Encontrando archivos de forma difícil

Mientras que el programa locate puede encontrar un archivos basándose solamente en su nombre, el programa find busca en un directorio (y sus subdirectorios) archivos por una variedad de atributos. Vamos a pasar mucho tiempo con find porque tiene un montón de características interesantes que veremos una y otra vez cuando empecemos a ver conceptos de programación en capítulos posteriores.

En su uso más simple, find necesita uno o más nombres de directorios para buscar. Por ejemplo, para producir una lista de nuestro directorio home:

```
[me@linuxbox ~]$ find ~
```

En la mayoría de las cuentas de usuario activas, producirá una gran lista. Como la lista se envía a la salida estándar, podemos desvíar la lista a otros programas. Usemos wc para contar el número de archivos:

```
[me@linuxbox ~]$ find ~ | wc -l
47068
```

¡Guau, hemos estado ocupados! La belleza de find es que puede usarse para identificar archivos que cumplen criterios específicos. Lo hace a través de aplicaciones (un poco extrañas) de *opciones*, *tests* y *acciones*. Veremos los tests primero.

Tests

Digamos que queremos una lista de directorios de nuestra búsqueda. Para hacerlo, podríamos añadir el siguiente test:

```
[me@linuxbox ~]$ find ~ -type d | wc -l
1695
```

Añadiendo el test -type d limitamos la búsqueda a directorios. Inversamente, podríamos haber limitado la búsqueda a archivos normales con este test:

```
[me@linuxbox ~]$ find ~ -type f | wc -l
38737
```

Aquí tenemos los test de tipos de archivo más comunes soportados por find:

Tabla 17-1: Tipos de archivo en find

Tipo de archivo	Descripción

b	Archivo especial de bloques de dispositivo
c	Archivo especial de carácter de dispositivo
d	Directorio
f	Archivo normal
l	Enlace simbólico

También podemos buscar por tamaño de archivo y nombre de archivo añadiendo algunos tests adicionales. Busquemos todos los archivos normales que coinciden con el patrón con comodín "*.JPG" y que sean mayores de un megabyte:

```
[me@linuxbox ~]$ find ~ -type f -name "*.JPG" -size +1M | wc -l
840
```

En este ejemplo, añadimos el test `-name` seguido del patrón con comodín. Fíjate como lo hemos incluido entre comillas para prevenir la expansión de nombre por el shell. A continuación, añadimos el test `-size` seguido de la cadena "+1M". El signo más al principio indica que estamos buscando archivos mayores que el número especificado. Un signo menos al principio cambiaría el significado de la cadena para que sea menor que el número especificado. No usar signo significa, "coincidencia del valor exacto". La letra final "M" indica que la unidad de medida es megabytes. Los siguientes caracteres pueden usarse para especificar unidades:

Tabla 17-2: Unidades de tamaño de find

Carácter	Unidad
b	Bloques de 512 bytes. Es la opción por defecto si no especificamos ninguna unidad.
c	Bytes
w	Palabras de 2 bytes
k	Kilobytes (unidades de 1024 bytes)
M	Megabytes (unidades de 1048576 bytes)
G	Gigabytes (unidades de 1073741824 bytes)

`find` soporta un gran número de tests diferentes. Abajo hay un paquete de los más comunes. Fíjate que en los casos en los que se requiere un argumento numérico, puede aplicarse la misma notación de "+" y "-" que vimos antes:

Tabla 17-3: Tests de find

Test	Descripción
`-cmin` *n*	Encuentra archivos o directorios cuyo contenido o atributos han sido modificado exactamente hace *n* minutos. Para especificar menos que *n* minutos, usa -*n* y para especificar más de *n* minutos usa +*n*

`-cnewer` *`archivo`*	Encuentra archivos o directorios cuyo contenido o atributos han sido modificados más recientemente que el archivo *archivo*.
`-ctime` *`n`*	Encuentra archivos o directorios cuyo contenido o atributos fueron modificados por última vez hace *`n*24`* horas.
`-empty`	Encuentra archivos y directorios vacíos
`-group` *`nombre`*	Encuentra archivos o directorios que pertenecen a *group*. *group* podría expresarse tanto como un nombre de grupo o una ID numérica de grupo.
`-iname` *`patrón`*	Como el test *-name* pero sensible a mayúsculas
`-inum` *`n`*	Encuentra archivos con el número de inodo *n*. Ésto es útil para encontrar todos los enlaces duros a un inodo particular.
`-mmin` *`n`*	Encuentra archivos o directorios cuyo contenido fue modificados por última vez hace *n* minutos.
`-mtime` *`n`*	Encuentra archivos o directorios cuyo contenido fue modificado por última vez hace *n*24* horas.
`-name` *`patrón`*	Encuentra archivos o directorios con la cadena `patrón`.
`-newer` *`archivo`*	Encuentra archivos y directorios cuyo contenido fue modificado más recientemente que el *`archivo`* especificado. Ésto es muy útil cuando escribimos scripts de hell que realizan copias de seguridad de archivos. Cada vez que haces una copia de seguridad, actualiza una archivo (como un log por ejemplo), y luego usa `find` para determinar que archivos fueron cambiados desde la última actualización.
`-nouser`	Encuentra archivos y directorios que no pertenecen a un usuario válido. Puede usarse para encontrar archivos pertenecientes a cuentas borradas o para detectar actividad de atacantes.
`-nogroup`	Encuentra archivos y directorios que no pertenecen a un grupo válido.
`-perm` *`modo`*	Encuentra archivos y directorios que tienen configuración de permisos en el *`modo`* especificado. *`modo`* podría expresarse tanto en octal como en notación simbólica.
`-samefile` *`nombre`*	Igual que el test *`-inum`*. Encuentra archivos que comparten el mismo número de inodo que el

	archivo *nombre*.
-size *n*	Encuentra archivos de tamaño *n*.
-type *c*	Encuentra archivos de tipo *c*.
-user *nombre*	Encuentra archivos o directorios que pertenezcan al usuario *nombre*. El usuario puede expresarse por un nombre de usuario o por un ID numérico de usuario.

Ésta no es una lista completa. La man page de `find` tiene todos los detalles.

Operadores

Incluso con todos los test que proporciona `find`, podríamos necesitar aún una mejor forma de describir las relaciones lógicas entre los test. Por ejemplo, ¿qué pasa si necesitamos determinar si todos los archivos y subdirectorios de un directorio tiene permisos seguros? Podríamos buscar todos los archivos con permisos que no son 0600 y los directorios con permisos que no son 0700. Afortunadamente, `find` cuenta con una forma de combinar test usando operadores lógicos para crear relaciones lógicas más complejas. Para expresar el test mencionado anteriormente, podríamos hacer ésto:

```
[me@linuxbox ~]$ find ~ \( -type f -not -perm 0600 \) -or \( -type
d
-not -perm 0700 \)
```

¡Uff! Seguro que te parece raro. ¿Qué es todo eso? Realmente, los operadores no son tan complicados una vez que los conoces. Aquí hay una lista:

Tabla 17-4: Operadores lógicos de find

Operador	Descripción
-and	Mira si los tests en ambas caras del operador son verdad. Puede abreviarse a `-a`. Fíjate que cuando no hay operador presente, `-and` está implícito por defecto.
-or	Comprueba si un test que está en algún lado del operador es falso. Puede abreviarse a `-o`.
-not	Comprueba si el test que sigue al operador es falso. Puede abreviarse con un signo de exclamación (!).
( )	Agrupa tests y operadores juntos para formar una expresión más grande. Se usa para controlar la precedencia de las evaluaciones lógicas. Por defecto, `find` evalua de izquierda a derecha. A menudo es necesario anular la precedencia por defecto para obtener el resultado deseado. Incluso si no es necesario, es útil a veces incluir los caracteres de agrupamiento para mejorar la legibilidad del comando. Fíjate que como los paréntesis tienen significado especial para el shell, deben ir entrecomillados cuando se usan en la línea de comandos para permitir que sean pasados como argumentos de `find`. Normalmente el carácter de la barra invertida se usa para ignorarlos.

Con esta lista de operadores a mano, deconstruyamos nuestro comando `find`. Cuando lo vemos desde el nivel superior, vemos que nuestros tests están ordenados como dos grupos separados por un operador `-or`:

(expresión 1) -or (expresión 2)

Esto tiene sentido, ya que estamos buscando archivos con una configuración de permisos determinados y directorios con una configuración distinta. Si estamos buscando tanto archivos como directorios, ¿por qué usamos `-or` en lugar de `-and`? Porque a medida que `find` busca a través de archivos y directorios, cada uno es evaluado para ver si coincide con los tests especificados. Queremos saber *si es* un archivo con malos permisos *o* un directorio con malos permisos. No puede ser los dos al mismo tiempo. Así que si expandimos las expresiones agrupadas, podemos verlo de esta forma:

```
( archivo con malos permisos ) -or ( directorio con malos permisos
)
```

Nuestro próximo desafío es como buscar “malos permisos” ¿Cómo lo hacemos? En realidad no lo haremos. Lo que buscaremos será “permisos no buenos”, ya que sabemos que son “permisos buenos”. En el caso de archivos, definimos bueno como 0600 y para directorios, 0700. La expresión que buscará archivos con permisos “no buenos” es:

```
-type f -and -not -perms 0600
```

y para directorios:

```
-type d -and -not -perms 0700
```

Cómo apuntamos en la tabla de operadores anterior, el operador `-and` puede eliminarse con seguridad, ya que está implícito por defecto. Así que si ponemos todo junto, tenemos nuestro comando definitivo:

```
find ~ ( -type f -not -perms 0600 ) -or ( -type d -not -perms 0700
)
```

Sin embargo, como los paréntesis tienen un significado especial para el shell, tenemos que “escaparlos” para prevenir que el shell trate de interpretarlos. Precediendo cada uno con una barra invertida conseguimos el truco.

Hay otra característica de los operadores lógicos que es importante entender. Digamos que tenemos dos expresiones separadas por un operador lógico:

```
expr1 -operador expr2
```

En todos los casos, *`expr1`* siempre se llevará a cabo; sin embargo, el operador determinará si *`expr2`* se realizará. Aquí tenemos cómo funciona:

Tabla 17-5: Lógica AND/OR de find

Resultado de expr1	Operador	expr2...

`Verdadero`	-and	se ejecuta siempre
`Falso`	-and	no se ejecuta
`Verdadero`	-or	no se ejecuta
`Falso`	-or	se ejecuta siempre

¿Por qué ocurre ésto? Para mejorar el rendimiento. Tomemos *`-and`*, por ejemplo. Sabemos que la expresión *`expr1 -and expr2`* no puede ser verdadera si el resultado de *`expr1`* es falso, luego es necesario que se ejecute *`expr2`*. Igualmente, si tenemos la expresión *`expr1 -or expr2`* y el resultado de expr1 es verdadero, no es necesario que se ejecute la *`expr2`* ya que ya sabemos que la expresión *`expr1 -or expr2`* es verdadera.

Ok, así que ayuda a que sea más rápido. ¿Por qué es importante? Es importante porque podemos contar con este comportamiento para controlar como se realizan las acciones, como veremos pronto.

Acciones predefinidas

¡Trabajemos! Tener una lista de resultados de nuestro comando `find` es útil, pero lo que realmente queremos hacer es actuar en los elementos de la lista. Afortunadamente, `find` permite que las acciones se realicen basándose en los resultados de búsqueda. Hay una serie de acciones predefinidas y varias formas de aplicar acciones definidas por el usuario. Primero, veamos algunas acciones predefinidas:

Tabla 17-6: Acciones predefinidas de find

Acción	**Descripción**
`-delete`	Borra el archivo buscado.
`-ls`	Realiza el equivalente a `ls -dils` en el archivo buscado. La salida se envía a la salida estándar.
`-print`	Envía la ruta completa del archivo buscado a la salida estándar. Es la acción por defecto si no se especifica ninguna otra acción.
`-quit`	Sale una vez que una búsqueda se ha realizado.

Como con los tests, hay muchas más acciones. Mira la man page de `find` para más detalles.

En nuestro primer ejemplo, hicimos esto:

```
find ~
```

que produjo una lista de cada archivo y subdirectorio contenidos en nuestro directorio home. Produjo una lista porque la acción `-print` está implícita si no se especifica ninguna otra acción. Luego nuestro comando podría expresarse así también:

```
find ~ -print
```

Podemos usar `find` para borrar archivos que cumplan ciertos criterios. Por ejemplo, para borrar

archivos que tengan la extensión “.BAK” (que a menudo se usa para designar archivos de copia de seguridad), podríamos usar este comando:

```
find ~ -type f -name '*.BAK' -delete
```

En este ejemplo, cada archivo en el directorio home del usuario (y sus subdirectorios) es buscado por si su nombre termina en .BAK. Cuando se encuentran, se borran.

Atención: No debería seguir sin decir ***que tengas extremo cuidado*** cuando uses la acción `-delete`. Siempre prueba el comando antes sustituyendo la acción `-print` por `-delete` para confirmar los resultados de búsqueda.

Antes de continuar, echemos otro vistazo a cómo afectan a las acciones los operadores lógicos. Considera el siguiente comando:

```
find ~ -type f -name '*.BAK' -print
```

Como hemos visto, este comando buscará cada archivo normal (`-type f`) cuyos nombres terminan en `.BAK (-name '*.BAK')` y enviará la ruta relativa de cada archivo coincidente a la salida estándar (`-print`). Sin embargo, la razón de que el comando funcione como lo hace viene determinada por la relación lógica entre cada test y su acción. Recuerda, hay, por defecto, una relación `-and` implícita entre cada test y cada acción. Podríamos expresar también el comando de esta forma para hacer la relación lógica más fácil de ver:

```
find ~ -type f -and -name '*.BAK' -and -print
```

Con nuestro comando expresado completamente, veamos a como afectan los operadores lógicos a su ejecución:

Test/Acción	Se ejecuta sólo si...
`-print`	`-type f` y `-name '*.BAK'` son verdaderos
`-name ‘*.BAK’`	`-type f` es verdadero
`-type f`	Se ejecuta siempre, ya que es el primer test/action en una relación `-and`.

Como la relación lógica entre los tests y las acciones determina cuales de ellos se ejecutan, podemos ver que el orden de los test y acciones es importante. Por ejemplo, si fuéramos a reordenar los test y las acciones para que la acción `-print` fuera la primera, el comando se comportaría de forma muy diferente:

```
find ~ -print -and -type f -and -name '*.BAK'
```

Esta versión del comando imprimirá cada archivo (la acción `-print` siempre se evalúa como verdadera) y luego busca por tipo de a archivo y la extensión de archivo especificada.

Acciones definidas por el usuario

Además de las acciones predefinidas, podemos invocar también comandos arbitrarios. La forma

tradicional de hacerlo es con la acción `-exec`. Esta acción funciona así:

```
-exec comando {} ;
```

donde *comando* es el nombre de un comando, {} es la representación simbólica de la ruta actual, y el punto y coma es un delimitador requerido indicando el final del comando. Aquí hay un ejemplo de usar `-exec` de forma que funcione como la acción `-delete` que vimos antes:

```
-exec rm '{}' ';'
```

De nuevo, como las llaves y el punto y coma tiene significado especial para el shell, deben ir entrecomillados o "escapados".

También es posible ejecutar una acción definida por el usuario interactivamente. Usando la acción `-ok` en lugar de `-exec`, el usuario es preguntado antes de la ejecución de cada comando especificado:

```
find ~ -type f -name 'foo*' -ok ls -l '{}' ';'
< ls ... /home/me/bin/foo > ? y
-rwxr-xr-x 1 me me 224 2007-10-29 18:44 /home/me/bin/foo
< ls ... /home/me/foo.txt > ? y
-rw-r--r-- 1 me me 0 2008-09-19 12:53 /home/me/foo.txt
```

En este ejemplo, buscamos archivos con nombres que empiecen con la cadena "foo" y ejecutamos el comando `ls -l` cada vez que encontremos uno. Usar la acción `-ok` pregunta al usuario antes de que se ejecute el comando `ls`.

Mejorando la eficiencia

Cuando usamos la acción `-exec`, arranca una nueva instancia del comando especificado cada vez que encuentra un archivo coincidente. Hay veces que podríamos preferir combinar todos los resultados de la búsqueda y arrancar una instancia única del comando. Por ejemplo, en lugar de ejecutar los comandos así:

```
ls -l archivo1
ls -l archivo2
```

podríamos preferir ejecutarlos así:

```
ls -l archivo1 archivo2
```

ésto hace que el comando se ejecute sólo una vez en lugar de muchas veces. Hay dos formas de hacer ésto. La forma tradicional, usando el comando externo `xargs` y la forma alternativa, usando una nueva característica del propio `find`. Hablaremos sobre la forma alternativa primero.

Cambiando el punto y coma del final por un signo más, activamos la capacidad de `find` de combinar los resultados de la búsqueda en una lista de argumentos para una ejecución única del comando deseado. Volviendo a nuestro ejemplo, ésto:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me me 224 2007-10-29 18:44 /home/me/bin/foo
```

```
-rw-r--r-- 1 me me 0 2008-09-19 12:53 /home/me/foo.txt
```

ejecutará `ls` cada vez que encuentre un archivo coincidente. Cambiando el comando a:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' +
-rwxr-xr-x 1 me me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me me 0 2008-09-19 12:53 /home/me/foo.txt
```

tendremos los mismos resultados, pero el sistema sólo habrá ejecutado el comando `ls` una vez.

xargs

El comando `xargs` realiza una función interesante. Acepta entradas de la entrada estándar y las convierte en una lista de argumentos para el comando especificado. Con nuestro ejemplo, podríamos usarlo así:

```
find ~ -type f -name 'foo*' -print | xargs ls -l
-rwxr-xr-x 1 me me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me me 0 2008-09-19 12:53 /home/me/foo.txt
```

Aquí vemos la salida del comando `find` entubada dentro de `xargs` lo que, sucesivamente, construye una lista de argumentos para el comando `ls` y luego lo ejecuta.

Nota: Mientras que el número de argumentos que pueden colocarse en una línea de comandos es bastante largo, no es ilimitado. Es posible crear comandos que son demasiado largos para que los acepte el shell. Cuando una línea de comandos excede la longitud máxima soportada por el sistema, `xargs` ejecuta el comando especificado con el máximo numero de argumentos posible y luego repite el proceso hasta que la entrada estándar esté exhausta. Para ver el tamaño máximo de la línea de comandos, ejecuta `xargs` con la opción `--show limits`.

Tratando con nombres de archivo simpáticos

Los sistemas como-Unix permiten espacios embebidos (¡e incluso nuevas líneas!) en los nombres de archivo. Esto causa problemas a programas como `xargs` que construyen listas de argumentos para otros programas. Un espacio embebido será tratado como un delimitador, y el comando resultante interpretará cada palabra separada por un espacio como un argumento separado. Para remediar ésto, `find` y `xarg` permiten el uso opcional de un carácter nulo como separador de argumentos. Un carácter nulo se define en ASCII como el carácter representado por el número cero (en lugar de, por ejemplo, el carácter espacio, que se define en ASCII como el carácter representado por el número 32). El comando `find` cuenta con la acción `-print0`, que produce una salida separada por nulos, y el comando xargs tiene la opción `--null`, que acepta entradas separadas por nulos. Aquí tenemos un ejemplo:

```
find ~ -iname '*.jpg' -print0 | xargs --null ls -l
```

Usando esta técnica, podemos asegurarnos que todos los archivos, incluso los que contienen espacios embebidos en sus nombres, son tratados correctamente.

Una vuelta al patio de juegos

Es hora de usar `find` de forma algo (al menos) práctica. Crearemos un patio de juegos y probaremos algo de lo que hemos aprendido.

Primero, creemos un patio de juegos con muchos subdirectorios y archivos:

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100}
[me@linuxbox ~]$ touch playground/dir-{001..100}/file-{A..Z}
```

¡Maravíllense con el poder de la línea de comandos! Con estas dos líneas, hemos creado un directorio patio de juegos que contiene 100 subdirectorios cada uno conteniendo 26 archivos vacíos. ¡Inténtalo con la GUI!

El método que hemos empleado para realizar esta magia incluye un comando familiar (`mkdir`), una exótica expansión de shell (llaves) y un nuevo comando, `touch`. Combinando `mkdir` con la opción `-p` (que hace que `mkdir` cree los directorios padres de las rutas especificadas) con expansión con llaves, podemos crear 100 subdirectorios.

El comando `touch` se usa normalmente para establecer o actualizar, el acceso, cambio y fecha de modificación de los archivos. Sin embargo, si el argumento del nombre de archivos es el de un archivo que no existe, se crea un archivo vacío.

En nuestro patio de juegos, hemos creado 100 instancias de un archivo llamado `file-A`. Encontremoslas:

```
[me@linuxbox ~]$ find playground -type f -name 'file-A'
```

Fijate que al contrario que `ls`, `find` no produce resultados ordenados. Su orden lo determina el dispositivo de almacenamiento. Podemos confirmar que realmente tenemos 100 instancias del archivo de esta forma:

```
[me@linuxbox ~]$ find playground -type f -name 'file-A' | wc -l
100
```

A continuación, veamos los archivos encontrados basándonos en su hora de modificación. Ésto será útil cuando creamos copias de seguridad o organizamos archivos en orden cronológico. Para hacerlo, primero crearemos un archivo de referencia contra el que compararemos la hora de modificación:

```
[me@linuxbox ~]$ touch playground/timestamp
```

Ésto crea un archivo vacío llamado `timestamp` y establece su hora de modificación en la hora actual. Podemos verificarlo usando otro comando útil, `stat`, que es un tipo de versión vitaminada de `ls`. El comando `stat` revela todo lo que el sistema entiende sobre un archivo y sus atributos:

```
[me@linuxbox ~]$ stat playground/timestamp
File: `playground/timestamp'
Size: 0 Blocks: 0 IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--) Uid: ( 1001/ me) Gid: ( 1001/ me)
Access: 2008-10-08 15:15:39.000000000 -0400
```

```
Modify: 2008-10-08 15:15:39.000000000 -0400
Change: 2008-10-08 15:15:39.000000000 -0400
```

Si “tocamos” el archivo de nuevo y luego lo examinamos con `stat`, veremos que las horas del archivo se han actualizado:

```
[me@linuxbox ~]$ touch playground/timestamp
[me@linuxbox ~]$ stat playground/timestamp
File: `playground/timestamp'
Size: 0 Blocks: 0 IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--) Uid: ( 1001/ me) Gid: ( 1001/ me)
Access: 2008-10-08 15:23:33.000000000 -0400
Modify: 2008-10-08 15:23:33.000000000 -0400
Change: 2008-10-08 15:23:33.000000000 -0400
```

A continuación, usemos `find` para actualizar algunos archivos de nuestro patio de juegos:

```
[me@linuxbox ~]$ find playground -type f -name 'file-B' -exec
touch
'{}' ';'
```

Ésto actualiza todos los archivos del patio de juegos llamados `file-B`. Luego usaremos `find` para identificar los archivos actualizados comparando todos los archivos con el archivo de referencia `timestamp`:

```
[me@linuxbox ~]$ find playground -type f -newer
playground/timestamp
```

El resultado contiene las 100 instancias del archivo `file-B`. Como realizamos un `touch` en todos los archivos del patio de juegos llamados `file-B` después de actualizar `timestamp`, ahora son más “nuevos” que `timestamp` y por lo tanto pueden ser identificados con el test `-newer`.

Finalmente, volvamos al test de malos permisos que realizamos anteriormente y apliquémoslo a `playground`:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 \) -or
\(-type d -not -perm 0700 \)
```

Este comando lista los 100 directorios y los 2600 archivos de `playground` (así como `timestamp` y el propio `playground`, haciendo un total de 2702) porque ninguno de ellos entra en nuestra definición de “buenos permisos”. Con nuestro conocimiento de los operadores y las acciones, podemos añadir acciones a este comando para aplicar nuevos permisos a los archivos y directorios en nuestro patio de juegos:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
chmod 0600 '{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod
0700 '{}' ';' \)
```

En el día a día, podríamos encontrar más fácil manejarnos con dos comandos, uno para los directorios y otro para los archivos, en lugar de este gran comando complejo, pero es bueno saber

que podemos hacerlo de esta forma. Lo importante aquí es entender cómo pueden usarse los operadores y las acciones juntos para realizar tareas útiles.

Opciones

Finalmente, tenemos las opciones. Las opciones se usan para controlar el alcance de una búsqueda con `find`. Podrían incluirse con otros test y acciones cuando se construyen expresiones con `find`. Aquí hay una lista de las más usadas:

Tabla 17-7: Opciones find

Opción	Descripción
`-depth`	Dirige `find` a procesar los archivos de un directorio antes del propio directorio. Esta opción se aplica automáticamente cuando se especifica la acción `-delete`.
`-maxdepth levels`	Establece el número máximo de niveles que `find` desciende en un árbol de directorios cuando realizamos tests y acciones.
`-mindepth levels`	Establece el número mínimo de niveles que `find` desciende en un árbol de directorios antes de aplicar tests y acciones.
`-mount`	Dirige `find` a no cruzar directorios que está montados en otros sistemas de archivos.
`-noleaf`	Dirige `find` a no optimizar su búsqueda asumiendo que está buscando en un sistema de archivos como-Unix. Ésto es necesario cuando buscamos en sistemas de archivos DOS/Windows y CD-ROMs

Resumiendo

Es fácil ver que `locate` es simple y que `find` es complicado. Ambos tienen sus usos. Tómate tiempo para explorar las muchas funciones de `find`. Puede, usándolo regularmente, mejora tu entendimiento de las operaciones de los sistemas de archivo Linux.

Para saber más

- Los programas `locate`, `updatedb`, `find` y `xargs` son parte del paquete GNU Project's *findutils*. El proyecto GNU ofrece un sitio web con una extensa documentación on-line, que es muy buena y deberías leerla si usas estos programas en entornos de alta seguridad: http://www.gnu.org/software/findutils/

Archivado y copias de seguridad

Una de las tareas primarias de un administrador de sistemas es mantener la seguridad de los datos del sistema. Una forma de hacerlo es realizando copias de seguridad periódicas de los archivos del sistema. Incluso si no eres el administrador del sistema, a menudo es útil hacer copias de las cosas y mover grandes colecciones de archivos de un lugar a otro y de un dispositivo a otro.

En este capítulo, veremos varios de los programas comunes usados para manejar colecciones de archivos. Están los archivos de compresión de archivos:

- `gzip` – Comprime y descomprime archivos
- `bzip2` – Un compresor ordenado por bloques

Los programas de archivado:

- `tar` – Utilidad de archivado en cinta
- `zip` – Empaqueta y comprime archivos

Y el programa de sincronización de archivos:

- `rsync` – Sincronización de archivos y directorios remotos

Comprimiendo archivos

A lo largo de la historia de la informática, ha existido una lucha por almacenar la mayor cantidad de datos posible en el mínimo espacio disponible, ya sea espacio en memoria, dispositivos de almacenamiento o ancho de banda. Muchos de los servicios de datos que hoy consideramos normales como reproductores de música portátiles, televisiones de alta definición o Internet de banda ancha, deben su existencia a las eficaces técnicas de *compresión de datos*.

La compresión de datos es el proceso de remover la *redundancia* de los datos. Consideremos un ejemplo imaginario. Digamos que tenemos una foto completamente negra con una dimensiones de 100 por 100 píxeles. En términos de almacenamiento de datos (asumiendo 24 bits, o 3 bytes por píxel), la imagen ocuparía 30.000 bytes de almacenamientos:

100 * 100 * 3 = 30.000

Una imagen que es toda de un color contiene datos completamente redundantes. Si fuéramos listos, podríamos codificar los datos de tal forma que simplemente describiría el hecho de que tenemos un bloque de 10.000 píxeles negros. Así que, en lugar de almacenar un bloque de datos que contenga 30.000 ceros (el negro se representa normalmente en archivos de imagen como cero), podríamos comprimir los datos al número 10.000, seguido de un cero para representar nuestros datos. Éste sistema de compresión de datos se llama *run-length enconding* y es una de las técnicas de compresión más rudimentarias. Las técnicas de hoy son mucho más avanzadas y complejas pero el objetivo básico sigue siendo el mismo, librarse de los datos redundantes.

Los *algoritmos de compresión* (las técnicas matemáticas usada para llevar a cabo la compresión) se clasifican en dos categorías, *lossless* (sin pérdida) y *lossy* (con pérdida). La compresión sin pérdida mantiene todos los datos contenidos en el original. Ésto significa que cuando un archivo es restaurado de una versión comprimida, el archivo restaurado es exactamente el mismo que la versión original sin comprimir. La compresión con pérdida, por otra parte, elimina datos al realizar la compresión, para permitir que se aplique una mayor compresión. Cuando se restaura un archivo con pérdida, no coincide con la versión original; en su lugar, es una aproximación muy parecida. Ejemplos de compresión con pérdida son JPG (para imágenes) y MP3 (para música). En este tema, veremos sólo la compresión sin pérdida, ya que la mayoría de los datos de un ordenador no pueden tolerar ninguna pérdida de datos.

gzip

El programa `gzip` se usa para comprimir uno o más archivos. Cuando lo ejecutamos, reemplaza el archivo original por una versión comprimida del original. El programa correspondiente `gunzip` se usa para restaurar archivos comprimidos a su original forma descomprimida. Aquí tenemos un ejemplo:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.*
```

```
-rw-r--r-- 1 me me 15738 2008-10-14 07:15 foo.txt
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me me 3230 2008-10-14 07:15 foo.txt.gz
[me@linuxbox ~]$ gunzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me me 15738 2008-10-14 07:15 foo.txt
```

En este ejemplo, creamos un archivo de texto llamado `foo.txt` desde un listado de directorio. A continuación, ejecutamos `gzip`, que reemplaza el archivo original con una versión comprimida llamada `foo.txt.gz`. En el listado de directorio de `foo.*`, vemos que el archivo original ha sido reemplazado con la versión comprimida, y la versión comprimida tiene más o menos la quinta parte del tamaño de la original. Podemos ver también que el archivo comprimido tiene los mismos permisos y datos de tiempo que el original.

A continuación, ejecutamos el programa `gunzip` para descomprimir el archivo. Tras ello, podemos ver que la versión comprimida del archivo ha sido reemplazada por la original, de nuevo con los permisos y datos de tiempos conservados.

`gzip` tiene muchas opciones. Aquí tenemos unas pocas:

Tabla 18-1: Opciones de gzip

Opción	Descripción
`-c`	Escribe la salida en la salida estándar y mantiene los archivos originales. También puede especificarse con `–-stdout` y `–-to-stdout`.
`-d`	Descomprime. Hace que gzip actúe como gunzip. También puede indicarse con `–-dcompress` o `–-uncompress.`
`-f`	Fuerza la compresión incluso si ya existe una versión comprimida del archivo original. También puede indicarse con `--force`.
`-h`	Muestra información de uso. También puede especificarse con `--help`.
`-l`	Lista las estadísticas de compresión para cada archivo comprimido. También puede especificarse con `--list`.
`-r`	Si uno o más argumentos de la línea de comandos son directorios, recursivamente comprime los archivos incluidos en ellos. También podría especificarse con `--recursive`.
`-t`	Prueba la integridad de un archivo comprimido. También podría especificarse con `--test`.
`-v`	Muestra mensajes mientras comprime. También puede indicarse con `--verbose`.
`-number`	Establece la cantidad de compresión. *number* es un entero en el rango entre 1 (más rápido, menos comprimido) y 9 (más lento, más comprimido). Los valores 1 y 9 también pueden expresarse como `--fast` y `--best`, respectivamente. El valor por defecto es 6.

Volviendo a nuestro ejemplo anterior:

```
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ gzip -tv foo.txt.gz
foo.txt.gz: OK
[me@linuxbox ~]$ gzip -d foo.txt.gz
```

Aquí, hemos reemplazado el archivo `foo.txt` con una versión comprimida llamada `foo.txt.gz`. A continuación, hemos probado la integridad de la versión comprimida, usando las opciones `-t` y `-v`. Finalmente hemos descomprimido el archivo de vuelta a su forma original.

`gzip` también puede usarse de formas interesantes vía entrada y salida estándar:

```
[me@linuxbox ~]$ ls -l /etc | gzip > foo.txt.gz
```

Este comando crea una versión comprimida de un listado de directorio.

El programa `gunzip`, que descomprime archivos gzip, asume que los nombres de archivo terminan con la extensión `.gz`, así que no es necesario especificarla, siempre que el nombre especificado no entre en conflicto con un archivo descomprimido existente.

```
[me@linuxbox ~]$ gunzip foo.txt
```

Si nuestro objetivo fuera sólo ver el contenido de un archivo de texto comprimido, podríamos hacer ésto:

```
[me@linuxbox ~]$ gunzip -c foo.txt | less
```

Alternativamente, hay un programa proporcionado con `gzip`, llamado `zcat`, que es equivalente a `gunzip` con la opción `-c`. Puede usarse como el comando `cat` en archivos comprimidos con `gzip`:

```
[me@linuxbox ~]$ zcat foo.txt.gz | less
```

Consejo: Hay un programa `zless` también. Realiza la misma función que el pipeline anterior.

bzip2

El programa `bzip2`, de Julian Seward, es similar a gzip, pero usa un algoritmo de compresión diferente que consigue niveles de compresión más altos a costa de la velocidad de compresión. En la mayoría de los aspectos, funciona de la misma forma que `gzip`. Un archivo comprimido con `bzip2` es marcado con la extensión `.bz2`:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-r--r-- 1 me me 15738 2008-10-17 13:51 foo.txt
[me@linuxbox ~]$ bzip2 foo.txt
[me@linuxbox ~]$ ls -l foo.txt.bz2
-rw-r--r-- 1 me me 2792 2008-10-17 13:51 foo.txt.bz2
[me@linuxbox ~]$ bunzip2 foo.txt.bz2
```

Como podemos ver, `bzip2` puede usarse de la misma forma que `gzip`. Todas las opciones

(excepto `-r`) que vimos para `gzip` también son soportadas en `bzip2`. Fíjate, sin embargo, que la opción del nivel de compresión (`-number`) tiene un significado algo diferente para `bzip2`. `bzip2` viene con `bunzip2` y `bzcat` para descomprimir los archivos.

`bzip2` también viene con el programa `bzip2recover`, que tratará de recuperar archivos `.bz2` dañados.

No seas un compresor compulsivo

Ocasionalmente veo gente tratando de comprimir un archivo, que ya ha sido comprimido con un algoritmo de compresión efectivo, haciendo algo así:

$ gzip picture.jpg

No lo hagas. ¡Probablemente sólo estarás malgastando tiempo y espacio! Si aplicas compresión a un archivo que ya ha sido comprimido, acabarás obteniendo un archivo más grande en realidad. Ésto es porque todas las técnicas de compresión incluyen un encabezado que se añade al archivo para describir la compresión. Si tratas de comprimir un archivo que ya no contiene información redundante, la compresión no proporcionará ningún ahorro para compensar el encabezado adicional.

Empaquetando archivos

Una tarea de gestión de archivos que se usa a menudo junto con la compresión es el *empaquetado*. Empaquetar es el proceso de recoger muchos archivos y empaquetarlos juntos en un gran archivo único. El empaquetado a menudo se hace como parte de los sistemas de copia de seguridad. También se usa cuando se mueven datos antiguos de un sistema a algún tipo de almacenamiento de largo plazo.

tar

En el mundo del software como-Unix, el programa `tar` es la herramienta clásica para empaquetar. Su nombre, abreviatura de *tape archive*, revela sus raíces como una herramienta para hacer copias de seguridad en cinta. Mientras que aún se usa para su tarea tradicional, está igualmente adaptada en otros dispositivos de almacenamiento.

A menudo vemos nombres de archivo que terminan con la extensión `.tar` o `.tgz`, que indican un paquete tar "plano" y un paquete comprimido con gzip, respectivamente. Un paquete tar puede consistir en un grupo de archivos separados, una o más jerarquía de directorios, o una mezcla de ambas. La sintaxis del comando funciona así:

```
tar modo[opciones] ruta...
```

donde modo es uno de los siguientes modos de operación (sólo se muestra una lista parcial; mira la man page de `tar` para ver la lista completa):

Tabla 18-2: Modos de tar

Modo	Descripción
`c`	Crea un paquete de una lista de archivos y/o directorios.

x	Extrae un archivo.
r	Añade rutas específicas al final de un archivo.
t	Lista el contenido de un paquete.

`tar` usa una forma un poco extraña para expresar las opciones, así que necesitaremos algunos ejemplos para mostrar como funciona. Primero, recreemos nuestro patio de juegos del capítulo anterior:

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100}
[me@linuxbox ~]$ touch playground/dir-{001..100}/file-{A..Z}
```

A continuación, creemos un paquete tar del patio de juegos completo:

```
[me@linuxbox ~]$ tar cf playground.tar playground
```

Este comando crea un paquete tar llamado `playground.tar` que contiene la jerarquía completa del directorio playground. Podemos ver que el modo y la opción `f`, que hemos usado para especificar el nombre del paquete `tar`, pueden ir juntas, y no requieren un guión delante. Fíjate, sin embargo, que el modo siempre debe ser especificado primero, antes que ninguna opción.

Para listar los contenidos del paquete, podemos hacer ésto:

```
[me@linuxbox ~]$ tar tf playground.tar
```

Para un listado más detallado, podemos añadir la opción `v` (verbose):

```
[me@linuxbox ~]$ tar tvf playground.tar
```

Ahora, extraigamos el patio de juegos en una nueva localización. Lo haremos creando un nuevo directorio llamado `foo`, cambiando el directorio y extrayendo el paquete tar:

```
[me@linuxbox ~]$ mkdir foo
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground.tar
[me@linuxbox foo]$ ls
playground
```

Si examinamos el contenido de `~/foo/playground`, vemos que el paquete ha sido instalado exitosamente, creando un reproducción precisa de los archivos originales. Hay una advertencia, sin embargo: A menos que estés operando como superusuario, los archivos y los directorios extraídos de paquetes toman la propiedad del usuario que realiza la restauración, en lugar del propietario original.

Otro comportamiento interesante de `tar` es la forma en que maneja las rutas en los archivos. Por defecto son relativas, en lugar de absolutas. `tar` hace ésto simplemente eliminando cualquier barra al principio de las rutas cuando creamos el archivo. Para demostralo, recrearemos nuestro paquete, esta vez especificando una ruta absoluta:

```
[me@linuxbox foo]$ cd
```

```
[me@linuxbox ~]$ tar cf playground2.tar ~/playground
```

Recuerda, `~/playground` se expandirá en `/home/me/playground` cuando pulsemos la tecla intro, así tendremos una ruta absoluta para nuestra demostración. A continuación, extraeremos el paquete como antes y veremos que ocurre:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar
[me@linuxbox foo]$ ls
home playground
[me@linuxbox foo]$ ls home
me
[me@linuxbox foo]$ ls home/me
playground
```

Aquí podemos ver que cuando extrajimos nuestro segundo paquete, volvió a crear el directorio `home/me/playground` relativo a nuestro directorio de trabajo actual, `~/foo`, no relativo al directorio root, como habría sido con rutas absolutas. Ésto podría parecer una forma extraña de trabajar, pero es en realidad más útil de esta manera, ya que nos permite extraer paquetes en cualquier sitio en lugar de ser forzados a extraerlos en sus localizaciones originales. Repitiendo el ejercicio con la inclusión de la opción verbose (`v`) nos dará una imagen más clara de lo que está pasando.

Consideremos un ejemplo hipotético, aunque práctico, de `tar` en acción. Imagina que queremos copiar el directorio home y su contenido de un sistema a otro y tenemos un gran disco duro USB que usamos para la transferencia. En nuestro sistema Linux moderno, la unidad se monta "automáticamente" en el directorio `/media`. Imaginemos también que el disco tiene nombre de volumen `BigDisk` cuando lo conectamos. Para crear el paquete tar, podemos hacer lo siguiente:

```
[me@linuxbox ~]$ sudo tar cf /media/BigDisk/home.tar /home
```

Después de que se escriba el paquete tar, desmontamos el disco y lo conectamos al segundo ordenador. De nuevo, se monta en `/media/BigDisk`. Para extraer el paquete, hacemos lo siguiente:

```
[me@linuxbox2 ~]$ cd /
[me@linuxbox2 /]$ sudo tar xf /media/BigDisk/home.tar
```

Lo que es importante ver aquí es que tenemos que cambiar primero el directorio a /, para que la extracción sea relativa al directorio root, ya todas las rutas dentro del paquete son relativas.

Cuando extraemos un paquete, es posible limitar que se extrae de él. Por ejemplo, si queremos extraer un archivo individual de un paquete, podría hacerse así:

tar xf paquete.tar ***ruta***

Añadiendo la *ruta* al final del comando, `tar` sólo restaurará el archivo especificado. Pueden especificarse múltiples rutas. Fíjate que la ruta deber ser tal y como está almacenada en el paquete. Cuando especificamos rutas, los comodines no están soportados normalmente; sin embargo, la versión GNU de `tar` (que es la versión que encontramos más a menudo en las distribuciones Linux) los soporta con la opción `--wilcards`. Aquí tenemos un ejemplo usando nuestro paquete `playground.tar` anterior:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar --wildcards
'home/me/pla
yground/dir-*/file-A'
```

Este comando extraerá sólo los archivos marcados con la ruta especificada incluyendo el comodín `dir-*`.

`tar` se usa a menudo en conjunción con `find` para producir paquetes. En este ejemplo, usaremos `find` para producir una serie de ficheros e incluirlos en un paquete:

```
[me@linuxbox ~]$ find playground -name 'file-A' -exec tar rf
playground.tar '{}' '+'
```

Aquí usamos `find` para buscar todos los archivos de `playground` llamados `file-A` y luego, usando la acción `-exec`, invocamos a `tar` en el modo inclusión `(r)` para añadir los archivos buscados al paquete `playground.tar`.

Usar `tar` con `find` es una buena forma de crear *copias incrementales* de un árbol de directorios o de un sistema completo. Usando `find` para buscar archivos más nuevos que un archivo con una marca de tiempo, podríamos crear una paquete que sólo contenga archivos más nuevos que el último archivo, asumiendo que el archivo con la marca de tiempo se ha actualizado justo antes de que se cree cada paquete.

`tar` puede también usar tanto entrada estándar como salida estándard. Aquí hay un ejemplo comprensible:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name 'file-A' | tar cf -
--filesfrom=- | gzip > playground.tgz
```

En este ejemplo, usamos el programa `find` para producir una lista de ficheros coincidentes y los entubamos en `tar`. Si el nombre de fichero "-" es especificado, toma el significado de entrada o salida estándar, según sea necesario (Por cierto, este acuerdo de usar "-" para representar entrada/salida estándar se usa en muchos otros programas también). La opción `--files-from` (que también puede especificarse como -T) hace que `tar` lea su lista de rutas de un archivo en lugar de de la línea de comandos. Finalmente, el paquete producido por `tar` se entuba en `gzip` para crear el paquete comprimido `playground.tgz`. La extensión `.tgz` es la extensión convencional dada a los paquetes tar comprimidos con gzip. La extensión `.tar.gz` también se usa a veces.

Aunque hemos usado el programa `gzip` externamente para producir nuestro paquete comprimido, las versiones modernas de `tar` GNU soportan tanto compresión gzip como bzip2 directamente, con el uso de las opciones z y j, respectivamente. Usando nuestros ejemplos previos como base, podemos simplificarlo de esta forma:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar czf
playground.tgz -T -
```

Si hubiéramos querido crear un paquete bzip comprimido, podríamos haber hecho ésto:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar cjf
playground.tbz -T -
```

Simplemente cambiando la opción de compresión de `z` a `j` (y cambiando la extensión del archivo de salida a `.tbz` para indicar que es un archivo comprimido con bzip2) activamos la compresión bzip2.

Otro uso interesante de la entrada y salida estándar con el comando `tar` implica transferir archivos entre sistemas a través de una red. Imagina que tenemos dos máquinas ejecutando un sistema como-Unix equipadas con `tar` y `ssh`. En tal escenario, podríamos transferir un directorio de un sistema remoto (llamado `remote-sys` para este ejemplo) a nuestro sistema local:

```
[me@linuxbox ~]$ mkdir remote-stuff
[me@linuxbox ~]$ cd remote-stuff
[me@linuxbox remote-stuff]$ ssh remote-sys 'tar cf - Documents' |
tar
xf -
me@remote-sys’s password:
[me@linuxbox remote-stuff]$ ls
Documents
```

Así podríamos copiar un documento llamado `Documents` de un sistema remoto `remote-sys` a un directorio incluido en el directorio llamado `remote-stuff` en el sistema local. ¿Cómo hacemos ésto? Primero, arrancamos el programa `tar` en el sistema remoto usando `ssh`. Notarás que `ssh` nos permite ejecutar un programa remotamente en un ordenador conectado a una red y "ver" el resultado en el sistema local - la salida estándar producida en el sistema el sistema remota se envía al sistema local para visualizarla. Podemos sacar ventaja de ésto haciendo a `tar` crear un paquete (el modo `c`) y enviarlo a la salida estándar, en lugar de un archivo (la opción `f` con el guión como argumento), de esta forma transportamos el archivo a través de un túnel encriptado proporcionado por `ssh` en el sistema local. En el sistema local, ejecutamos `tar` y lo hacemos expandir un paquete (el modo x) suministrado desde la entrada estándar (de nuevo con la opción `f` con el guión como argumento).

zip

El programa `zip` es tanto una herramienta de compresión como de empaquetado. El formato de archivo usado por el programa es familiar para los usuarios de Windows, ya que lee y escribe archivos .`zip`. En Linux, sin embargo, `gzip` es el programa de compresión predominante con `bzip2` muy cerca en segundo lugar.

En su uso más básico, `zip` se invoca así:

```
zip opciones archivozip archivo...
```

Por ejemplo, para hacer un archivo zip de nuestro patio de juegos, haríamos ésto:

```
[me@linuxbox ~]$ zip -r playground.zip playground
```

A menos que incluyamos la opción `-r` para recursividad, sólo el directorio `playground` (pero no

su contenido) se almacenará. Aunque la extensión .zip se añade automáticamente, la indicaremos por claridad.

Durante la creación del archivo zip, zip normalmente mostrará una serie de mensajes como éstos:

```
adding: playground/dir-020/file-Z (stored 0%)
adding: playground/dir-020/file-Y (stored 0%)
adding: playground/dir-020/file-X (stored 0%)
adding: playground/dir-087/ (stored 0%)
adding: playground/dir-087/file-S (stored 0%)
```

Estos mensajes muestran el estado de cada archivo añadido al paquete. zip añadirá archivo al paquete usando uno de estos dos métodos de almacenamiento: O "almacenará" un archivo sin compresión, como hemos visto aquí, o "reducirá" el archivo realizando compresión. El valor numérico mostrado tras el método de almacenamiento indica la cantidad de compresión realizada. Como nuestro patio de juegos sólo contiene archivos vacíos, no se realizará ninguna compresión de su contenido.

Extraer el contenido de un zip es sencillo cuando usamos el programa unzip:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip
```

Una cosa a tener en cuenta sobre zip (al contrario que en tar) es que si se especifica un archivo existente, se actualiza en vez de reemplazarse. Ésto significa que el archivo existente se conserva, pero los archivos nuevos se añaden y los coincidentes se reemplazan.

Los archivos pueden listarse y extraerse selectivamente de un archivo zip especificándolo a unzip:

```
[me@linuxbox ~]$ unzip -l playground.zip playground/dir-087/file-Z
Archive: ../playground.zip
Length Date Time Name
-------- ---- ---- ----
0 10-05-08 09:25 playground/dir-087/file-Z
-------- -------
0 1 file
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip playground/dir-
087/file-Z
Archive: ../playground.zip
replace playground/dir-087/file-Z? [y]es, [n]o, [A]ll, [N]one,
[r]ename: y
extracting: playground/dir-087/file-Z
```

Usar la opción -l hace que unzip sólo liste el contenido de un paquete sin extraer nada. Si no se especifican archivos, unzip listará todos los archivos del paquete. La opción -v puede añadirse para aumentar la información del listado. Fijate que cuando la extracción del paquete entra en conflicto con una archivo existente, el ususario es preguntado antes de que el archivo sea reemplazado.

Como tar, zip puede hacer uso de la entrada y la salida estándar, aunque su implementación no es tan útil. Es posible entubar una lista de nombres de archivo a zip con la opción -@:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name "file-A" | zip -@ file-
A.zip
```

Aquí usamos `find` para generar una lista de archivos coincidentes con el test `-name "file-A"`, y luego entubar la lista en `zip`, lo que crea el paquete `file-A.zip` conteniendo los archivos seleccionados.

`zip` también soporta escribir su salida a la salida estándar, pero su uso es limitado porque muy pocos programas pueden hacer uso de la salida. Desafortunadamente, el programa `unzip` no acepta entrada estándar. Ésto impide que `zip` y `unzip` se usen juntos para realizar copiado de archivos en red como `tar`.

`zip` puede, sin embargo, aceptar entrada estándar, así que puede usarse para comprimir la salida de otros programas:

```
[me@linuxbox ~]$ ls -l /etc/ | zip ls-etc.zip -
adding: - (deflated 80%)
```

En este ejemplo entubamos la salida de `ls` en `zip`. Como `tar`, `zip` interpreta el guión final como "usa la entrada estándar para el archivo de entrada."

El programa `unzip` permite que su salida se envíe a la salida estándar cuando se especifica la opción -p (para entubar):

```
[me@linuxbox ~]$ unzip -p ls-etc.zip | less
```

Hemos visto algunas de las cosas básicas que puede hacer `zip/unzip`. Ambos tienen un montón de opciones que añadir a su flexibilidad, aunque algunas son específicas para plataformas de otros sistemas. Las man pages de `zip` y `unzip` son muy buenas y contienen ejemplos útiles. Sin embargo el uso principal de estos programas es el intercambio de archivos con sistemas Windows, en lugar de comprimir y empaquetar en Linux, donde `tar` y `gzip` son claramente preferidos.

Sincronizando archivos y directorios

Una estrategia común para mantener una copia de seguridad de un sistema implica mantener uno o más directorios sincronizados con otro directorio (o directorios) localizados en el sistema local (normalmente un dispositivo de almacenamiento removible de algún tipo) o en un sistema remoto. Podríamos, por ejemplo, tener una copia local de un sitio web en desarrollo y sincronizarlo de vez en cuando con la copia "live" en un servidor web remoto.

En el mundo como-Unix, la herramienta preferida para esta tarea es `rsync`. Este programa puede sincronizar tanto directorios locales como remotos usando el *protocolo rsync remote-update*, que permite que `rsync` detecte rápidamente las diferencias entre los dos directorios y realizar una cantidad mínima de copiado para sincronizarlos. Ésto hace a `rsync` muy rápido y económico de usar, comparado con otros tipos de programas de copia.

`rsync` se invoca así:

```
rsync opciones origen destino
```

donde origen y destino son uno de los siguientes:

- Un archivo o directorio local
- Un archivo o directorio remoto en la forma *[usuario@]host:ruta*
- Un servidor rsync remoto especificado con una URI de *rsync://[usuario@]host[:puerto]/ruta*

Fíjate que o el origen o el destino deben ser un archivo local. La copia remoto-a-remoto no está soportada.

Probemos `rsync` en archivos locales. Primero, limpiemos nuestro directorio `foo`:

```
[me@linuxbox ~]$ rm -rf foo/*
```

A continuación, sincronizaremos el directorio `playground` con su copia correspondiente en `foo`:

```
[me@linuxbox ~]$ rsync -av playground foo
```

Hemos incluido tanto la opción `-a` (para archivar - que produce recursividad y conservación de los atributos de los archivos) y la opción `-v` (salida verbose) para hacer un espejo del directorio `playground` dentro de `foo`. Mientras el comando se ejecuta, veremos una lista de los archivos y directorios que se están copiando. Al final, veremos un mensaje de sumario como este:

```
sent 135759 bytes received 57870 bytes 387258.00 bytes/sec
total size is 3230 speedup is 0.02
```

indicando la cantidad de copia realizada. Si ejecutamos el comando de nuevo, veremos un resultado diferente:

```
[me@linuxbox ~]$ rsync -av playgound foo
building file list ... done
sent 22635 bytes received 20 bytes 45310.00 bytes/sec
total size is 3230 speedup is 0.14
```

Fíjate que no había listado de archivos. Ésto es porque `rsync` detectó que no había diferencias entre `~/playground` y `~/foo/playground`, y por tanto no necesitaba copiar nada. Si modificamos un archivo en `playground` y ejecutamos `rsync` de nuevo:

```
[me@linuxbox ~]$ touch playground/dir-099/file-Z
[me@linuxbox ~]$ rsync -av playground foo
building file list ... done
playground/dir-099/file-Z
sent 22685 bytes received 42 bytes 45454.00 bytes/sec
total size is 3230 speedup is 0.14
```

vemos que `rsync` detectó el cambio y copió sólo el archivo actualizado.

Como ejemplo práctico, consideremos el disco duro externo imaginario que usamos anteriormente con `tar`. Si conectamos el disco a nuestro sistema y, de nuevo, se monta en `/media/BigDisk`, podemos realizar una útil copia de seguridad del sistema creando primero un directorio llamado `/backup` en el disco externo, y luego usando `rsync` para copiar lo más importante de nuestro

sistema en el disco externo:

```
[me@linuxbox ~]$ mkdir /media/BigDisk/backup
[me@linuxbox ~]$ sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup
```

En este ejemplo, hemos copiado los directorios `/etc`, `/home`, y `/usr/local` de nuestro sistema a nuestro dispositivo de almacenamiento imaginario. Hemos incluido la opción `--delete` para eliminar archivos que existieran en el dispositivo de copia y que ya no existieran en el dispositivo original (ésto es irrelevante la primera vez que hacemos una copia de seguridad, pero será útil en sucesivas copias). Repitiendo el procedimiento de conectar el disco externo y ejecutar este comando `rsync` sería una forma útil (aunque no ideal) de mantener un pequeño sistema respaldado. De acuerdo, un alias sería muy útil aquí, también: Podríamos crear un alias y añadirlo a nuestro archivo `.bashrc` para tener la utilidad:

```
alias backup='sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup'
```

Ahora, todo lo que tenemos que hacer es conectar nuestro disco externo y ejecutar el comando `backup` para realizar el trabajo.

Usando rsync a través de una red

Uno de los verdaderos encantos de `rsync` es que puede usarse para copiar archivos a través de una red. Después de todo, la "r" de `rsync` significa "remoto". La copia remota puede hacerse de dos formas. La primera es con otro sistema que tenga `rsync` instalado, a través de un programa shell remoto como `ssh`. Digamos que tenemos otro sistema en nuestra red local con mucho espacio disponible en el disco duro y queremos realizar una operación de copia de seguridad usando el sistema remoto en lugar de un disco externo. Asumiendo que ya hay un directorio llamado `/backup` donde podríamos depositar nuestros archivos, podríamos hacer ésto:

```
[me@linuxbox ~]$ sudo rsync -av --delete --rsh=ssh /etc /home
/usr/local remote-sys:/backup
```

Hemos hecho dos cambios en nuestro comando para facilitar el copiado a través de la red. Primero, hemos añadido la opción `--rsh=ssh`, que le dice a `rsync` que use el programa `ssh` como su shell remoto. De esta forma, podemos usar un tunel ssh encriptado para transferir datos con seguridad desde el sistema local hasta el host remoto. Segundo, hemos especificado el host remoto añadiendo su nombre (en este caso el host remoto se llama `remote-sys`) como prefijo de la ruta de destino.

La segundo forma en que podemos usar `rsync` para sincronizar archivos a través de una red es usando un *servidor rsync*. `rsync` pude configurarse para funcionar como un demonio y escuchar solicitudes remotas de sincronización. Ésto se hace a menudo para realizar copias espejo de un sistema remoto. Por ejemplo, Red Hat Software mantiene un gran repositorio de paquetes de software en desarrollo para su distribución Fedora. Es útil para testeadores de software replicar esta colección durante la fase de testeo del ciclo de liberación de la distribución. Como los archivos en el repositorio cambian con frecuencia (a menudo más de una vez al día), es recomendable mantener una copia espejo local con sincronización periódica, en lugar de copiar todo el repositorio. Uno de esos repositorios está alojado en Georgia Tech; podríamos clonarlo usando nuestra copia local de `rsync` y su servidor rsync así:

```
[me@linuxbox ~]$ mkdir fedora-devel
[me@linuxbox ~]$ rsync -av -delete
rsync://rsync.gtlib.gatech.edu/fed
ora-linux-core/development/i386/os fedora-devel
```

En este ejemplo, hemos usado la URI del servidor rsync remoto, que consiste en un protocolo (`rsync://`), seguido del nombre de host remoto (`rsync.gtlib.gatech.edu`), seguido de la ruta del repositorio.

Resumiendo

Hemos viso los programas más comunes de compresión y empaquetado en Linux y otros sistemas operativos como-Unix. Para empaquetar archivos, la combinación `tar/gzip` es el método preferido en sistemas como-Unix mientras que `zip/unzip` se usa para interoperar con sistemas Windows. Finalmente, hemos visto el programa `rsync` (mi favorito personal) que es muy útil para sincronizar eficientemente archivos y directorios entres sistemas.

Para saber más

- Las man pages de todos los comandos vistos aquí son muy claras y contienen ejemplos útiles. Además, el proyecto GNU tiene un buen manual online para su versión de `tar`. Puede encontrarse aquí:
  http://www.gnu.org/software/tar/manual/index.html

Expresiones regulares

En los próximos capítulos, vamos a ver herramientas que se usan para manejar texto. Como hemos visto, los datos de texto juegan un papel importante en todos los sistemas como-Unix, por ejemplo Linux. Pero antes de poder apreciar al completo todas las funciones ofrecidas por estas herramientas, tenemos que examinar primero una tecnología que está frecuentemente asociada a los usos más sofisticados de éstas herramientas - las *expresiones regulares*.

Según hemos ido navegando por las muchas funciones y soluciones ofrecidas por la línea de comandos, hemos encontrado algunas funciones y comandos verdaderamente antiguos, como la expansión del shell y el entrecomillado, los atajos de teclado y el historial de comandos, no hace falta mencionar al editor vi. Las expresiones regulares continúan esta "tradición" y pueden ser (podría decirse) la función más arcaica de todas ellas. Esto no quiere decir que el tiempo que nos lleve aprender sobre ellas no merezcan el esfuerzo. Justo lo contrario. Un buen entendimiento nos permitirá usarlas para realizar cosas impresionantes, aunque su verdadero valor no sea aparente inicialmente.

¿Qué son las expresiones regulares?

Simplemente digamos que, las expresiones regulares son notaciones simbólicas usadas para identificar patrones en el texto. En muchos aspectos, se parecen al método de los comodines del shell para encontrar archivos y rutas, pero en una escala mucho más grande. Las expresiones regulares están soportadas por muchas herramientas de la línea de comandos y por la mayoría de los lenguajes de programación para facilitar la solución de problemas de manipulación de texto. Sin embargo, para confundir más las cosas, no todas las expresiones regulares con iguales; varían un poco de herramienta a herramienta y de lenguaje de programación a lenguaje de programación. Para

nuestro tema, nos limitaremos a expresiones regulares como se describen en el estándar POSIX (que cubrirá la mayoría de las herramientas de la línea de comandos), al contrario que muchos lenguajes de programación (principalmente *Perl*), que usa un catálogo algo más grande y rico de notaciones.

grep

El principal programa que usaremos para trabajar con expresiones regulares es nuestro antiguo colega, `grep`. El nombre "grep" deriva realmente de la frase "global regular expresión print - impresión global de expresiones regulares", luego podemos ver que `grep` tiene algo que ver con las expresiones regulares. En esencia, `grep` busca en archivos de texto la coincidencia con una expresión regular especificada y produce líneas que contengan una coincidencia a la salida estándar.

Hasta ahora, hemos usado `grep` con cadenas concretas, así:

```
[me@linuxbox ~]$ ls /usr/bin | grep zip
```

Esto listará todos los archivos en el directorio `/usr/bin` cuyos nombres contengan la cadena "zip".

El programa `grep` acepta opciones y argumentos de esta forma:

```
grep [opciones] regex [archivo...]
```

donde *regex* es una expresión regular.

Aquí tenemos una lista de las opciones más comunes usadas en `grep`:

Tabla 20-1: Opciones de grep

Opción	Descripción
`-i`	Ignora las mayúsculas. No distingue entre caracteres mayúscula y minúscula. También puede indicarse como `--ignore-case`.
`-v`	Coincidencia inversa. Normalmente, `grep` muestra líneas que contienen un patrón. Esta opción hace que `grep` muestra todas las líneas que no contengan un patrón. También puede indicarse como `--invert-match`.
`-c`	Muestra el número de coincidencias (o no-coincidencias si también se especifica la opción `-v`) en lugar de las propias líneas. También puede indicarse como `--count`.
`-l`	Muestra el nombre de cada archivo que contenga un patrón en lugar de las propias líneas. También puede indicarse como `--files-witth-matches`.
`-L`	Como la opción `-l`, pero muestra sólo los nombres de los archivos que no contengan el patrón. Puede indicarse también como `--files-without-match`.
`-n`	Precede cada línea coincidente con el número de la línea dentro del archivo. También puede indicarse como `--line-number`.
`-h`	Para búsquedas multi-archivo, suprime los nombres de archivo de la salida. También puede indicarse como `--no-filename.`

Para explorar más a fondo `grep`, creemos algunos archivos de texto para buscar:

```
[me@linuxbox ~]$ ls /bin > dirlist-bin.txt
[me@linuxbox ~]$ ls /usr/bin > dirlist-usr-bin.txt
[me@linuxbox ~]$ ls /sbin > dirlist-sbin.txt
[me@linuxbox ~]$ ls /usr/sbin > dirlist-usr-sbin.txt
[me@linuxbox ~]$ ls dirlist*.txt
dirlist-bin.txt dirlist-sbin.txt dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

Podemos realizar una búsqueda simple de nuestra lista de archivos así:

```
[me@linuxbox ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

En este ejemplo, `grep` busca todos los archivos listados por la cadena `bzip` y encuentra dos coincidencias, ambos en el archivo `dirlist-bin.txt`. Si sólo estuviéramos interesados en la lista de archivos que contienen las coincidencias, podríamos especificar la opción `-l`:

```
[me@linuxbox ~]$ grep -l bzip dirlist*.txt
dirlist-bin.txt
```

Inversamente, si quisiéramos sólo ver una lista de los archivos que no contienen un patrón, podríamos hacer ésto:

```
[me@linuxbox ~]$ grep -L bzip dirlist*.txt
dirlist-sbin.txt
dirlist-usr-bin.txt

dirlist-usr-sbin.txt
```

Metacaracteres y literales

Aunque no parezca aparente, nuestras búsquedas con `grep` han usado expresiones regulares todo el tiempo, aunque muy simples. La expresión regular "bzip" significa que una coincidencia ocurrirá sólo si la línea del archivo contiene al menos cuatro caracteres y que en algún lugar de la línea los caracteres "b", "z", "i" y "p" se encuentran en ese orden, sin otros caracteres en medio. Los caracteres en la cadena "bzip" son *caracteres literales* que coinciden consigo mismos. Además de los literales, las expresiones regulares pueden también incluir *metacaracteres* que se usan para especificar coincidencias más complejas. Los metacaracteres para las expresiones regulares son los siguientes:

```
^ $ . [ ] { } - ? * + ( ) | \
```

Todos los demás caracteres se consideran literales, aunque la barra invertida se utiliza en algunos casos para crear *meta secuencias*, así como para que los metacaracteres puedan ser "escapados" y tratados como literales en lugar de ser interpretados como metacaracteres.

Nota: Como podemos ver, muchos metacaracteres de expresiones regulares son también caracteres que tienen significado para el shell cuando se realiza la expansión. Cuando pasamos expresiones

regulares que contienen metacaracteres en la línea de comandos, es vital que estén entrecomillados para prevenir que el shell trate de expandirlos.

El carácter cualquiera

El primer metacaracter que veremos es el punto, que se usa para buscar cualquier carácter. Si lo incluimos en una expresión regular, encontrará cualquier carácter en esa posición. Aquí tenemos un ejemplo:

```
[me@linuxbox ~]$ grep -h '.zip' dirlist*.txt
bunzip2
bzip2
bzip2recover
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

Hemos buscado cualquier linea en nuestro archivos que coincidan con la expresión regular ".zip". Hay un par de cosas interesantes a tener en cuenta en los resultados. Fíjate que el programa `zip` no ha sido encontrado. Es porque la inclusión del metacaracter punto en nuestra expresión regular incrementa la longitud de la coincidencia requerida a cuatro caracteres, y como el nombre "zip" sólo contiene tres, no coincide. También, si algún archivo en nuestra lista contiene la extensión `.zip`, debería haber coincidido también, porque el punto en la extensión del archivo es tratado como "cualquier carácter" también.

Anclas

El símbolo de intercalación `(^)` y el signo del dolar `($)` se tratan como *anclas* en la expresiones regulares. Esto significa que hacen que la coincidencia ocurra sólo si la expresión regular se encuentra al principio de la línea `(^)` o al final de la línea `($)`:

```
[me@linuxbox ~]$ grep -h '^zip' dirlist*.txt
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
[me@linuxbox ~]$ grep -h 'zip$' dirlist*.txt
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
unzip
```

```
zip
[me@linuxbox ~]$ grep -h '^zip$' dirlist*.txt
zip
```

Aquí hemos buscado en la lista de archivos si la cadena "zip" se encuentra al principio de línea, al final de la línea o en una línea donde está tanto al principio como al final de la línea (p.ej., que él mismo sea la línea). Fíjate que la expresión '^$' (un principio y un final con nada en medio) encontrará líneas en blanco.

Un ayudante de crucigramas

Incluso con nuestro limitado conocimiento de las expresiones regulares en este momento, podemos hacer algo útil.

A mi esposa le encantan los crucigramas y a veces me pide ayuda con alguna pregunta concreta. A veces como, "¿Que palabra de cinco letras cuya tercera letra es "j" y la ultima letra es "r" significa...?" Este tipo de preguntas me hacen pensar. ¿Sabes que tu sistema Linux tiene un diccionario? Lo tiene. Echa un vistazo al directorio `/usr/share/dict` y encontrarás uno, o varios. Los archivos del diccionario localizados allí son sólo largas listas de palabras, una por línea, ordenadas alfabéticamente. En mi sistema, el archivo `words` contiene alrededor de 98.500 palabras.
Para encontrar posibles respuestas a la pregunta del crucigrama anterior, podríamos hacer ésto:

```
[me@linuxbox ~]$ grep -i '^..j.r$' /usr/share/dict/words
Major
major
```

Usando la expresión regular, podemos encontrar todas las palabras del archivo del diccionario que tienen cinco letras y tienen una "j" en la tercera posición y una "r" en la última posición.

Expresiones entre corchetes y clases de caracteres

Además de encontrar un carácter en una determinada posición en nuestra expresión regular, podemos también encontrar un único carácter de una colección específica de caracteres usando *expresiones entre corchetes*. Con las expresiones entre corchetes, podemos especificar una colección de caracteres (incluyendo caracteres que en otro casos serían interpretados como metacaracteres) para que sean encontrados. En este ejemplo, usamos una colección de dos caracteres:

```
[me@linuxbox ~]$ grep -h '[bg]zip' dirlist*.txt
bzip2
bzip2recover
gzip
```

encontramos cualquier línea que contenga la cadena "bzip" o "gzip".

Una colección puede contener cualquier número de caracteres, y los metacaracteres perder su significado especial cuando se coloquen entre corchetes. Sin embargo, hay dos casos en que los metacaracteres se usan dentro de las expresiones entre corchetes, y tienen significados diferentes. El primero es el símbolo de intercalación (^), que se usa para indicar negación; el segundo es el guión

(-), que se usa para indicar un rango de caracteres.

Negación

Si el primer carácter en una expresión entre corchetes es el símbolo de intercalación (^), los caracteres restantes se toman como una colección de caracteres que no deben estar presentes en la posición dada. Podemos hacer esto modificando nuestro ejemplo anterior:

```
[me@linuxbox ~]$ grep -h '[^bg]zip' dirlist*.txt
bunzip2
gunzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

Con la negación activada, obtenemos una lista de archivos que contienen la cadena "zip" precedida de cualquier carácter excepto "b" o "g". Fíjate que el archivo `zip` no ha sido encontrado. Una configuración de negación de carácter requiere todavía un carácter en la posición dada, pero el carácter no debe ser un miembro de la lista de negaciones.

El símbolo de intercalación sólo implica negación si es el primer carácter dentro de una expresión entre corchetes; de otra forma, pierde su significado especial y pasa a ser un carácter ordinario en la lista.

Rangos de caracteres tradicionales

Si queremos construir una expresión regular que encuentre cualquier archivo en nuestra lista que empiece con una letra mayúscula, podríamos hacer ésto:

```
[me@linuxbox ~]$ grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXZY]'
dirlist*.txt
```

Es simplemente una forma de poner todas las letras mayúsculas en una expresión entre corchetes. Pero la idea de escribir todo eso es muy preocupante, así que aquí tenemos otra forma:

```
[me@linuxbox ~]$ grep -h '^[A-Z]' dirlist*.txt
MAKEDEV
ControlPanel
GET
HEAD
POST
X
X11
Xorg
MAKEFLOPPIES
NetworkManager
NetworkManagerDispatcher
```

Usando un rango de tres caracteres, podemos abreviar las 26 letras. Cualquier rango de caracteres puede expresarse de esta forma incluyendo rangos múltiples, como esta expresión que encuentra todos los archivos cuyo nombre empieza con letras y números:

```
[me@linuxbox ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

En los rangos de caracteres, vemos que el carácter guión se trata especialmente, entonces ¿cómo incluimos el carácter guión en una expresión entre corchetes? Haciéndolo el primer carácter en la expresión. Considera estos dos ejemplos:

```
[me@linuxbox ~]$ grep -h '[A-Z]' dirlist*.txt
```

Ésto encontrará todos los archivos cuyo nombre contiene una letra mayúscula. Mientras que:

```
[me@linuxbox ~]$ grep -h '[-AZ]' dirlist*.txt
```

encontrará todos los archivos cuyo nombre contiene un guión, o una letra "A" mayúscula o una letra "Z" mayúscula.

Clases de caracteres POSIX

Los rangos tradicionales de caracteres son una forma fácilmente compresible y efectiva de manejar el problema de especificar colecciones de caracteres rápidamente. Desafortunadamente, no siempre funcionan. Aunque no hemos encontrado problemas usando `grep` hasta ahora, podríamos tener problemas con otros programas.

Volviendo al capítulo 4, vimos como los comodines se usan para realizar expansiones en los nombres de archivos. En dicho tema, dijimos que los rangos de caracteres podían usarse de forma casi idéntica a la forma en que se usan en expresiones regulares, pero aquí está el problema:

```
[me@linuxbox ~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

(Dependiendo de la distribución Linux, obtendremos una lista diferente de archivos, posiblemente una lista vacía. Este ejemplo es de Ubuntu). Este comando produce el resultado esperado - una lista de los archivos cuyos nombre comienzan con una letra mayúscula, pero:

```
[me@linuxbox ~]$ ls /usr/sbin/[A-Z]*
/usr/sbin/biosdecode
/usr/sbin/chat
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/chroot
/usr/sbin/cleanup-info
/usr/sbin/complain
/usr/sbin/console-kit-daemon
```

con este comando obtenemos un resultado completamente diferente (sólo se muestra una lista parcial de resultados). ¿Por qué? Es una larga historia, pero aquí tienes un resumen:

En la época en que Unix fue desarrollado por primera vez, sólo entendía caracteres ASCII, y esta característica refleja este hecho. En ASCII, los primeros 32 caracteres (los números 0 a 31) son códigos de control (cosas como tabuladores, espacios y retornos de carro). Los siguientes 32 (32 a 63) contienen caracteres imprimibles, incluyendo la mayoría de los signos de puntuación y los números del cero al nueve. Los siguientes 32 (64 a 95) contienen las letras mayúsculas y algunos signos de puntuación más. Los últimos 31 (números del 96 al 127) contienen las letras mayúsculas y todavía más signos de puntuación. Basándose en este ordenamiento, los sistemas que usan ASCII utilizan un *orden de colación* que tiene esta pinta:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

Esto difiere del orden correcto del diccionario, que es así:

```
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
```

Cuando la popularidad de Unix se expandió fuera de los Estados Unidos, creció la necesidad de soportar caracteres que no existen en Inglés U.S. La tabla ASCII se incrementó para usar ochos bits completos, añadiendo los números de caracteres 128-255, que albergan muchos más idiomas. Para soportar esta capacidad, los estándares POSIX introdujeron un concepto llamado un *local*, que puede ajustarse para seleccionar la configuración de caracteres necesarias para una localización particular. Podemos ver la configuración de idioma de nuestro sistema usando este comando:

```
[me@linuxbox ~]$ echo $LANG
en_US.UTF-8
```

Con esta configuración, las aplicaciones que cumplen con POSIX usarán el orden del diccionarios en lugar del orden ASCII. Ésto explica el comportamiento de los comandos anteriores. Un rango de caracteres de `[A-Z]` cuando es interpretado en orden del diccionario incluye todos los caracteres alfabéticos excepto la "a" minúscula, de ahí nuestros resultados.

Para evitar este problema, el estándar POSIX incluye un número de clases de caracteres que proporcionan rango útiles de caracteres. Está descrito en la siguiente tabla:

Tabla 19-2: Clases de caracteres POSIX

Clases de Caracteres	**Descripción**
`[:alnum:]`	Los caracteres alfanuméricos. En ASCII, equivalente a: `[A-Za-z0-9]`
`[:word:]`	Los mismo que `[:alnum:]`, con el añadido del carácter subrayado (_).
`[:alpha:]`	Los caracteres alfabéticos. En ASCII, equivalente a: `[a-Za-z]`
`[:blank:]`	Incluye los caracteres del espacio y tabulador.
`[:cntrl:]`	Los caracteres de control ASCII. Incluyen los caracteres ASCII del 0 al 31 y 127.
`[:digit:]`	Los números del cero al nueve.
`[:graph:]`	Los caracteres visibles. En ASCII, incluye los caracteres del 33 al 126.
`[:lower:]`	Las letras minúsculas.

`[:punct:]`	Los símbolos de puntuación. En ASCII, equivalente a: `[-!"#$%&'()*+,./:;<=>?@[\\\]_`{|}~]`
`[:print:]`	Los caracteres imprimibles. Los caracteres de `[:graph:]` más el carácter espacio.
`[:space:]`	Los caracteres de espacio en blanco, incluyendo el espacio, el tabulador, el salto de carro, nueva linea, tabulador vertical, y salto de página. En ASCII equivalente : `[ \t\r\n\v\f]`
`[:upper:]`	Los caracteres mayúsculas.
`[:xdigit:]`	Los caracteres usados para expresar números hexadecimales. En ASCII, equivalente a: `[0-9A-Fa-f]`

Incluso con las clases de caracteres, sigue sin haber una forma conveniente de expresar rangos parciales, como [A-M].

Usando las clases de caracteres, podemos repetir nuestro listado de directorio y ver un resultado mejorado:

```
[me@linuxbox ~]$ ls /usr/sbin/[[:upper:]]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

Recuerda, sin embargo, que esto no es un ejemplo de expresión regular, es por el contrario el resultado de una expansión de ruta de shell. Lo vemos porque las clases de caracteres POSIX pueden usarse en ambos casos.

Volviendo al orden tradicional

Puedes optar porque tu sistema use el orden tradicional (ASCII) cambiando el valor de la variable de entorno `LANG`. Como vimos antes, la variable `LANG` contiene el nombre del idioma y catálogo de caracteres usados en configuración local. Este valor fue determinado originalmente cuando seleccionaste un idioma de instalación cuando tu Linux fue instalado.

Para ver la configuración local, usa el comando `locale`:

```
[me@linuxbox ~]$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
```

```
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

Para cambiar el local para usar el comportamiento tradicional de Unix, cambia la variable `LANG` a `POSIX`:

```
[me@linuxbox ~]$ export LANG=POSIX
```

Fíjate que este cambio convierte tu sistema a Inglés U.S. (más específicamente, ASCII) en su catálogo de caracteres, así que asegúrate que es realmente lo que quieres.

Puedes hacer que este cambio sea permanente añadiendo esta línea a tu archivo `.bashrc`:

```
export LANG=POSIX
```

POSIX básico vs. Expresiones regulares extendidas

Justo cuando pensábamos que no podía ser más confuso, descubrimos que POSIX también divide las expresiones regulares en dos tipos: *expresiones regulares básicas (BRE - Basic regular expressions)* y *expresiones regulares extendidas (ERE - extended regular expressions)*. Las funciones que hemos visto hasta ahora son soportadas por cualquier aplicación que sea compatible con POSIX e implemente BRE. Nuestro programa `grep` es uno de esos programas.

¿Cuál es la diferencia entre BRE y ERE? Es un asunto de metacaracteres. Con BRE, se reconocen los siguientes metacaracteres:

```
^ $ . [ ] *
```

Todos los demás caracteres se consideran literales. Con ERE, se añaden los siguientes metacaracteres (y sus funciones asociadas):

```
( ) { } ? + |
```

Sin embargo (y esta es la parte divertida), los caracteres "(", ")", "{" y "}" se tratan como metacaracters en BRE si son escapados con una barra invertida, mientras que con ERE, preceder cualquier metacaracter con una barra invertida hace que sea tratado como un literal. Cualquier rareza que se presente será tratada en los temas siguientes.

Como las funciones que vamos a ver a continuación son parte de ERE, vamos a necesitar un `grep` diferente. Tradicionalmente, se ha utilizado el programa `egrep`, pero la versión GNU de `grep` también soporta expresiones regulares cuando se usa la opción `-E`.

POSIX

Durante los años ochenta, Unix se volvió un sistema operativo comercial muy popular, pero alrededor de 1988, el mundo Unix se volvió muy confuso. Muchos fabricantes de ordenadores licenciaron el código fuente Unix de sus creadores, AT&T, y vendieron varias versiones del sistema operativo con sus equipos. Sin embargo, en su esfuerzo de crear diferenciación de producto, cada fabricante añadió cambios y extensiones propietarias. Ésto empezó a limitar la compatibilidad del

software. Como siempre pasa con los vendedores propietarios, cada uno intentaba un juego ganador de "atrapar" a sus clientes. La época oscura de la historia de Unix es conocida hoy en día como "*la Balcanización*".

Aparece el IEEE (Institute of Electrical and Electronics Engineers - Instituto de Ingenieros Eléctricos y Electrónicos). A medidados de los 80, el IEEE comenzó a desarrollar una serie de estándares que definieran como se comportan los sistemas Unix (y como-Unix). Estos estándares, formalmente conocidos como IEEE 1003, definen las *application programing interfaces - Interfaces de programación de aplicaciones* (APIs), shell y utilidades que deben encontrarse en un sistema como-Unix estándar. El nombre "POSIX", que viene de *Portable Operating System Interface* - *Interfaz de Sistema Operativo Portable* (con la "X" al final para que tenga más chispa), fue sugerido por Richard Stallman (si, ese Richard Stallman), y fué adoptado por el IEEE.

Alternancia

La primera característica de las expresiones regulares que veremos se llama *alternancia*, que es la función que nos permite que una coincidencia ocurra de entre una serie de expresiones. Justo como una expresión entre llaves que permite que un carácter único se encuentre dentro de una serie de caracteres especificados, la alternancia permite coincidencias dentro de una serie de cadenas y otras expresiones regulares.

Para demostrarlo, usaremos `grep` junto con `echo`. Primero, probemos una antigua coincidencia sencilla:

```
[me@linuxbox ~]$ echo "AAA" | grep AAA
AAA
[me@linuxbox ~]$ echo "BBB" | grep AAA
[me@linuxbox ~]$
```

Un ejemplo muy sencillo, en el que entubamos la salida de `echo` dentro de `grep` y vemos el resultado. Cuando ocurre una coincidencia, vemos que se imprime; cuando no hay coincidencias, no vemos resultados.

Ahora añadiremos alternancia, indicada por el metacaracter de la barra vertical:

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB'
AAA
[me@linuxbox ~]$ echo "BBB" | grep -E 'AAA|BBB'
BBB
[me@linuxbox ~]$ echo "CCC" | grep -E 'AAA|BBB'
[me@linuxbox ~]$
```

Aquí vemos la expresión regular `'AAA|BBB'`, que significa "encuentra la cadena AAA o la cadena BBB." Fíjate que como es una característica extendida, añadimos la opción `-E` a `grep` (aunque podríamos simplemente usar el programa `egrep` en su lugar), e incluiremos la expresión regular entre comillas para prevenir que el shell interprete el metacaracter de la barra vertical como el operador de entubar:

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB|CCC'
AAA
```

Para combinar la alternancia con otros elementos de expresiones regulares, podemos usar () para

separar la alternancia:

```
[me@linuxbox ~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

Esta expresión encontrará los nombres de archivo en nuestras listas que empiecen con "bz", "gz" o "zip". Si eliminamos los paréntesis, el significado de esta expresión regular:

```
[me@linuxbox ~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

cambia y encuentra cualquier nombre de archivo que comience por "bz" o *contenga* "gz" o "zip".

Cuantificadores

Las expresiones regulares extendidas soportan varias formas de especificar el número de veces que se encuentra un elemento.

? - Encuentra un elemento cero u una vez

Este cuantificador significa, en la práctica, "Haz el elemento precedente opcional." Digamos que queremos comprobar la validez de un número de teléfono y consideramos que un número de teléfono es válido si encaja en alguna de estas dos formas:

(nnn) nnn-nnnn
nnn nnn-nnnn

donde "*n*" es un numeral. Podríamos construir una expresión regular como esta:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

En esta expresión, seguimos a los caracteres de los paréntesis con signos de interrogación para indicar que tienen que ser comprobados cero o una vez. De nuevo, como los paréntesis son normalmente metacaracteres (en ERE), los precedemos con barras invertidas para hacer que sean tratadas como literales.

Probémoslo:

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9]
[0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-
9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
555 123-4567
[me@linuxbox ~]$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-
9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
[me@linuxbox ~]$
```

Aquí vemos que la expresión encuentra ambos formatos del número de teléfono, pero no encuentra ninguno que contenga caracteres no numéricos.

* - Encuentra un elemento cero o más veces

Como el metacaracter ?, el * se usa para señalar un elemento opcional; sin embargo, al contrario que ?, el elemento puede ocurrir cualquier número de veces, no sólo una. Digamos que queremos ver si una cadena era una frase; o sea, que comienza con una mayúscula, luego contiene cualquier número de letras mayúsculas o minúsculas y espacios, y termina con un punto. Para encontrar esta definición (muy básica) de una frase, podríamos usar una expresión regular como ésta:

```
[[:upper:]][[:upper:][:lower:] ]*\.
```

La expresión consta de tres elementos: un expresión entre corchetes conteniendo la clase de caracteres [:upper:], una expresión entre corchetes conteniendo tanto la clase de caracteres [:upper:] como [:lower:] y un espacio, y un punto escapado por una barra invertida. El segundo elemento está precedido de un metacaracter *, de forma que tras la letra mayúscula del principio de nuestra frase, cualquier cantidad de letras mayúculas o minúsculas y espacios que le sigan serán encontrados:

```
[me@linuxbox ~]$ echo "This works." | grep -E '[[:upper:]]
[[:upper:][:lower:] ]*\.'
This works.
[me@linuxbox ~]$ echo "This Works." | grep -E '[[:upper:]]
[[:upper:][:lower:] ]*\.'
This Works.
[me@linuxbox ~]$ echo "this does not" | grep -E '[[:upper:]]
[[:upper:][:lower:] ]*\.'
[me@linuxbox ~]$
```

La expresión coincide en los dos primeros test, pero no en el tercero, ya que carece de la mayúscula al principio y del punto al final.

+ - Encuentra un elemento una o más veces

El metacaracter + funciona de forma muy parecida a *, excepto que requiere al menos una instancia del elemento precedente para que ocurra una coincidencia. Aquí tenemos una expresión regular que sólo encontrará líneas consistentes en grupos de uno o más caracteres alfabéticos separados por un espacio:

```
^([[:alpha:]]+ ?)+$
[me@linuxbox ~]$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'
This that
[me@linuxbox ~]$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'
a b c
[me@linuxbox ~]$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$ echo "abc d" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$
```

Vemos que esta expresión no encuentra la línea "a b 9", porque contiene un carácter no alfabético; ni encuentra "abc d", por que los caracteres "c" y "d" están separados por más de un espacio en blanco.

{} - Encuentra un elemento un número específico de veces

Los metacaracteres { y } se usan para expresar el número mínimo y máximo de coincidencias requeridas. Pueden especificarse de cuatro formas diferentes:

Tabla 19-3: Especificando el número de coincidencias

Especificador	Significado
{*n*}	Encuentra el elemento precedente y ocurre exactamente *n* veces.
{*n*,*m*}	Encuentra el elemento precedente y ocurre al menos *n* veces, pero no más de *m* veces.
{*n*,}	Encuentra el elemento precedente y ocurre *n* o más veces.
{,*m*}	Encuentra el elemento precedente y ocurre no más de *m* veces.

Volviendo a nuestro ejemplo anterior con los números de teléfono, podemos usar este método de especificar repeticiones para simplificar nuestra expresión regular de:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

a:

```
^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$
```

Probémoslo:

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9]{3}\)?
[0-9]{3}-[0-9]{4}$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9]{3}\)?
[0-9]{3}-[0-9]{4}$'
555 123-4567
[me@linuxbox ~]$ echo "5555 123-4567" | grep -E '^\(?[0-9]{3}\)?
[0-9]{3}-[0-9]{4}$'
[me@linuxbox ~]$
```

Como podemos ver, nuestra expresión revisada puede validar correctamente tanto los números con paréntesis como sin paréntesis, mientras que rechaza aquellos números que no están formateados correctamente.

Poniendo las expresiones regulares a trabajar

Veamos algunos comandos que ya sabemos para ver como pueden usarse con expresiones regulares.

Validando una lista de teléfonos con grep

En nuestro ejemplo anterior, vimos como comprobábamos si un número telefónico estaba correctamente formateado. Un escenario más realista sería chequear una lista de números en su lugar, así que hagamos una lista. Haremos ésto recitando un encantamiento mágico a la línea de comandos. Será mágico porque no hemos visto la mayoría de los comandos involucrados, pero no te preocupes. Los veremos en próximos capítulos. Aquí está el encantamiento:

```
[me@linuxbox ~]$ for i in {1..10}; do echo "(${RANDOM:0:3}) $
{RANDO
M:0:3}-${RANDOM:0:4}" >> phonelist.txt; done
```

Este comando producirá un archivo llamado `phonelist.txt` que contiene diez números de teléfono. Cada vez que repetimos el comando, otros diez números se añaden a la lista. También podemos cambiar el valor `10` que está cerca del principio del comando para producir más o menos números de teléfono. Si examinamos el contenido del archivo, sin embargo, vemos que tenemos un problema:

```
[me@linuxbox ~]$ cat phonelist.txt
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
```

Algunos números están mal formateados, lo que es perfecto para nuestro propósito, ya que usaremos `grep` para validarlos.

Un método de validación útil sería escanear un archivo para encontrar números no válidos y mostrar la lista resultante en pantalla:

```
[me@linuxbox ~]$ grep -Ev '^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$'
phonelist.txt
(292) 108-518
(129) 44-1379
[me@linuxbox ~]$
```

Aquí usamos la opción `-v` para producir la coincidencia inversa ya que sólo mostraremos líneas de la lista que no coincidan con la expresión especificada. La propia expresión incluye metacaracteres ancla en los extremos para asegurarnos que el número no tiene caracteres extra al final. Esta expresión también requiere que estén presentes los paréntesis en un número válido, al contrario que nuestro número de teléfono del ejemplo anterior.

Encontrando nombres de archivos feos con find

El comando `find` soporta un test basado en un expresión regular. Hay una consideración importante a tener en cuenta cuando usamos expresiones regulares con `find` en lugar de `grep`. Mientras que `grep` imprimirá una linea cuando la línea *contiene* una cadena que coincide con una expresión, `find` requiere que la ruta *coincida exactamente* con la expresión regular. En el siguiente ejemplo, usaremos `find` con una expresión regular para encontrar cada ruta que contenga un carácter que no sea miembro de la siguiente lista:

```
[-_./0-9a-zA-Z]
```

Un escaneo como este revelará rutas que contenga espacios y otros potenciales caracteres ofensivos:

```
[me@linuxbox ~]$ find . -regex '.*[^-_./0-9a-zA-Z].*'
```

Debido al requerimiento de que coincida exactamente la ruta completa, usamos `.*` en ambos extremos de la expresión para buscar cero o más instancias de cada carácter. En el centro de la expresión, usamos una expresión entre corchetes negada conteniendo nuestra colección de caracteres de ruta aceptables.

Buscando archivos con locate

El programa `locate` soporta expresiones regulares tanto básicas (la opción `--regexp`) como extendidas (la opción `--regex`). Con él, podemos realizar muchas de las mismas operaciones que hicimos antes con nuestros archivos `dirlist`:

```
[me@linuxbox ~]$ locate --regex 'bin/(bz|gz|zip)'
/bin/bzcat
/bin/bzcmp
/bin/bzdiff
/bin/bzegrep
/bin/bzexe
/bin/bzfgrep
/bin/bzgrep
/bin/bzip2
/bin/bzip2recover
/bin/bzless
/bin/bzmore
/bin/gzexe
/bin/gzip
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

Usando alternancia, realizamos un búsqueda de rutas que contengan `bin/bz`, `bin/gz` o `/bin/zip`.

Buscando texto con less y vim

`less` y `vim` comparten el mismo método de búsqueda de texto. Pulsando la tecla / seguida de una expresión regular realizaremos una búsqueda. Si usamos `less` para ver nuestro archivo `phonelist.txt`:

```
[me@linuxbox ~]$ less phonelist.txt
```

y luego buscamos por nuestra expresión de validación:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
/^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

`less` resaltará las cadenas que coinciden, dejando las que no valen fáciles de eliminar:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
(END)
```

`vim`, por otra parte, soporta expresiones regulares básicas, de forma que nuestra expresión aparecería así:

```
/([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}
```

Podemos ver que la expresión es prácticamente igual; sin embargo, muchos de los caracteres que se consideran metacaracteres en expresiones extendidas se consideran literales en expresiones básicas. Sólo se tratan como metacaracteres cuando los escapamos con una barra invertida. Dependiendo de la configuración particular de vim en nuestro sistema, la coincidencia será resaltada. Si no, prueba este comando de modos:

```
:hlsearch
```

para activar la búsqueda resaltada.

Nota: Dependiendo de tu distribución, vim soportará o no la búsqueda resaltada. Ubuntu, en particular, contiene una versión muy simplificada de vim por defecto. En ese tipo de sistemas, puedes usar tu administrador de paquetes para instalar la versión completa de vim.

Resumiendo

En este capítulo, hemos visto algunos de los muchos usos de las expresiones regulares. Podemos encontrar incluso más si usamos expresiones regulares para buscar aplicaciones opcionales que las usen. Podemos hacerlo buscando las man pages:

```
[me@linuxbox ~]$ cd /usr/share/man/man1
[me@linuxbox man1]$ zgrep -El 'regex|regular expression' *.gz
```

El programa `zgrep` proporciona una interfaz para `grep`, permitiéndonos leer archivos comprimidos. En nuestro ejemplo, buscamos la sección uno de la man page comprimida en su localización usual. El resultado de este comando es una lista de archivos que contienen tanto la cadena "regex" como "regular expression". Como podemos ver, las expresiones regulares muestran un montón de programas.

Hay una función en las expresiones regulares básicas que no hemos visto. Se llaman *retroreferencias*, esta función será tratada en el siguiente capítulo.

Para saber más

Hay muchos recursos online para aprender expresiones regulares, incluyendo varios tutoriales y chuletas.

Además, la Wikipedia tiene buenos artículos en los siguientes enlaces:

- POSIX: http://en.wikipedia.org/wiki/Posix
- ASCII: http://en.wikipedia.org/wiki/Ascii

Procesado de texto

Todos los sistemas operativos como-Unix dependen en gran medida de los archivos de texto para varios tipos de almacenamiento de datos. Así que tiene sentido que haya muchas herramientas para manipular texto. En este capítulo, veremos programas que se usan para "cocinar" texto. En el siguiente capítulo, veremos más procesado de texto, centrándonos en programas que se utilizan para formatear el texto para imprimirlo y otros tipos de consumo humano.

Este capítulo revisitará a algunos viejos amigos y nos presentará algunos nuevos:

- `cat` - Encadena archivos e imprime en la salida estándar
- `sort` - Ordena líneas en archivos de texto
- `uniq` - Reporta u omite líneas repetidas
- `cut` - Elimina secciones de cada línea de archivos
- `paste` - Une lineas de archivos
- `join` - Une líneas de dos archivos en un campo común
- `comm` - Compara dos archivos ordenados línea por línea
- `diff` - Compara archivos línea por línea
- `patch` - Aplica un archivo de diferencia a uno original
- `tr` - Traduce o borra caracteres
- `sed` - Editor en continuo para filtrar y transformar texto
- `aspell` - Comprobador de deletreo interactivo

Aplicaciones de texto

Hasta ahora, hemos aprendido un par de editores de texto (`nano` y `vim`), hemos visto un puñado de archivos de configuración, y hemos presenciado la salida de docenas de comandos, todo en texto. Pero ¿para qué más se usa el texto? Resulta que para muchas cosas.

Documentos

Mucha gente escribe documentos usando formatos de texto plano. Mientras que es fácil ver como un archivo de texto pequeño puede ser útil para guardar notas simples, también es posible escribir grandes documentos en formato texto. Un enfoque popular es escribir un documento de texto largo en formato texto y luego usar un *lenguaje de marcas* para describir el formato del documento final. Muchos papers científicos están escritos usando este método, ya que los sistemas de procesado de texto basados en Unix fueron casi los primeros sistemas que soportaron diseños de tipografía avanzada necesarios para los escritores de disciplinas técnicas.

Páginas web

El tipo de documento electrónico más popular del mundo es probablemente la página web. Las páginas web son documentos de texto que usan *HTML (Hypertext Markup Language - Lenguaje de marcas de hipertexto)* o *XML (Extensible Markup Language - Lenguaje de marcas extensible)* como lenguajes de marcas para describir el formato visual de los documentos.

Email

El email es un medio intrínsecamente basado en texto. Incluso los adjuntos que no son texto son convertidos a una representación textual para transmitirlos. Podemos ver esto nosotros mismos descargando un mensaje de email y viéndolo con `less`. Veremos que el mensaje comienza con un *encabezado* que describe la fuente del mensaje y el procesado que ha recibido durante su viaje, seguido por un *cuerpo* del mensaje con su contenido.

Salida de impresora

En sistemas como-Unix, la salida destinada para una impresora se manda como texto plano o, si la página contiene gráficos, se convierte en un formato de texto en *lenguaje de descripción de página* llamado *PostScript*, que se envía a un programa que genera los puntos gráficos a imprimir.

Código fuente de programas

Muchos de los programas de la línea de comandos que encontramos en los sistemas como-Unix fueron creados para soportar administración de sistemas y desarrollo de software, y los programas de procesado de texto no son una excepción. Muchos de ellos están diseñados para resolver problemas de desarrollo de software. La razón por la que el procesado de texto es importante para los desarrolladores de software es que todo software nace como un texto. El *código fuente*, la parte del programa que el programador escribe en realidad, siempre está en formato texto.

Revisitando antiguos amigos

Volviendo al Capítulo 6 (Redirección), aprendimos algunos comandos que sirven para aceptar entrada estándar además de argumentos de línea de comandos. Sólo los vimos brevemente entonces, pero ahora le echaremos un vistazo más de cerca para ver como pueden usarse para realizar procesado de texto.

cat

El programa cat tiene un gran número de opciones interesantes. Muchas de ellas se usan para ayudarnos a visualizar mejor el contenido del texto. Un ejemplo es la opción -A, que se usa para mostrar caracteres no imprimibles en el texto. Hay veces que queremos saber si los caracteres de control están incrustados en nuestro texto visible. Los más comunes son los caracteres de tabulación (en lugar de espacios) y retornos de carro, a menudo presentes como caracteres de fin de línea en archivos de texto estilo MS-DOS. Otra situación común es un archivo que contiene líneas de texto con espacios al final.

Creemos un archivo de texto usando cat como un procesador de texto primitivo. Para hacerlo, sólo introduciremos el comando cat (sólo especificando un archivo para redirigir la salida) y escribiremos nuestro texto, seguido de Enter para terminar la línea apropiadamente, luego Ctrl-d para indicar a cat que hemos alcanzado el final del archivo. En este ejemplo, introducimos un carácter tabulador al principio y continuamos la línea con algunos espacios al final:

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox jumped over the lazy dog.
[me@linuxbox ~]$
```

A continuación, usaremos cat con la opción -A para mostrar el texto:

```
[me@linuxbox ~]$ cat -A foo.txt
^IThe quick brown fox jumped over the lazy dog. $
[me@linuxbox ~]$
```

Como podemos ver en los resultados, el carácter tabulador en nuestro texto se representa como ^I. Es una notación común que significa "Control-I" que, resulta que es lo mismo que el carácter tab. También vemos que aparece un $ al final real de la línea, indicando que nuestro texto contiene espacios al final.

cat también tiene opciones que se usan para modificar texto. Las dos más prominentes son -n, que numera las líneas, y -s, que suprime la salida de líneas en blanco múltiples. Podemos demostrarlo así:

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox
jumped over the lazy dog.
[me@linuxbox ~]$ cat -ns foo.txt
1 The quick brown fox
2
3 jumped over the lazy dog.
[me@linuxbox ~]$
```

En este ejemplo, hemos creado una nueva versión de nuestro archivo de texto foo.txt, que contiene dos líneas de texto separadas por dos líneas en blanco. Tras procesarlo con cat con las opciones -ns, la línea en blanco extra se elimina y el resto de líneas son numeradas. Aunque no es un proceso muy grande a realizar sobre el texto, es un proceso.

Texto MS-DOS Vs. Texto Unix

Una de las razones por las que querrías usar `cat` para buscar caracteres no imprimibles en el texto es para eliminar retornos de carro ocultos. ¿De donde vienen los los retornos de carro ocultos? ¡De DOS y Windows! Unix y DOS no definen el final de una linea de la misma forma en los archivos de texto. Unix finaliza una línea con un carácter salto de línea (ASCII 10) mientras que MS-DOS y sus derivados usan la secuencia del retorno de carro (ASCII 13) y salto de línea para terminar cada línea de texto.

Hay varias formas de convertir archivos de formato DOS a UNIX. En muchos sistemas Linux, hay programas llamados `dos2unix` y `unix2dos`, que pueden convertir archivos de texto de y hacia formato DOS. Sin embargo, si no tienes `dos2unix` en tu sistema, no te preocupes. El proceso de convertir texto de formato DOS a Unix es muy simple; simplemente implica eliminar los retornos de carro innecesarios. Esto lo realizan fácilmente fácilmente un par de programas que veremos más tarde en éste capítulo.

sort

El programa `sort` ordena el contenido de la entrada estándar, o de uno o más archivos especificados en la línea de comandos, y manda el resultado a la salida estándar. Usando la misma técnica que usamos con `cat`, podemos demostrar el procesado de la entrada estándar directamente desde el teclado:

```
[me@linuxbox ~]$ sort > foo.txt
c
b
a
[me@linuxbox ~]$ cat foo.txt
a
b
c
```

Tras introducir el comando, escribimos las letras "c", "b" y "a", seguida de nuevo de `Ctrl-d` para indicar el final del archivo. Entonces vemos el archivo resultante y vemos que las líneas ahora aparecen ordenadas.

Como `sort` puede aceptar múltiples archivos en la línea de comandos como argumentos, es posible unir múltiples archivos en un único total. Por ejemplo, si tenemos tres archivos de texto y queremos combinarlos en un único archivo ordenado, podríamos hacer algo como ésto:

```
sort file1.txt file2.txt file3.txt > final_sorted_list.txt
```

`sort` tiene varias opciones interesantes. Aquí hay una lista parcial:

Tabla 20-1: Opciones comunes de ordenamiento

Opción	Opción larga	Descripción
-b	--ignore-leading-blanks	Por defecto, el ordenado se realiza en la línea completa, empezando con el primer carácter de la línea. Esta opción hace que sort ignore los espacios al principio de la

		línea y calcula el ordenado basándose en el primer carácter que no es un espacio en blanco en la línea.
`-f`	`--ignore-case`	Hace el ordenamiento sensible a mayúsculas.
`-n`	`--numeric-sort`	Realiza el ordenado basado en la evaluación numérica de una cadena. Usar esta opción permite que el ordenado se realice según valores numéricos en lugar de valores alfabéticos.
`-r`	`--reverse`	Ordena inversamente. Los resultados son descendentes en lugar de ascendentes.
`-k`	`--key=`*`campo1`*`[,`*`campo2`*`]`	Orden basado en un campo clave localizado entre *campo1* y *campo2* en lugar de en la línea entera. Veremos este tema luego.
`-m`	`--merge`	Trata cada argumento como el nombre de un archivo preordenado. Une múltiples archivos en uno sólo ordenado resultante sin realizar ningún ordenamiento adicional.
`-o`	`--output=`*`archivo`*	Envía salida ordenada a *archivo* en lugar de a la salida estándar.
`-t`	`--fiel-separator=`*`carácter`*	Define el caracter separador de campos. Por defecto los campos se separan por espacios o tabuladores.

Aunque la mayoría de las opciones vistas antes son muy autoexplicativas, algunas no lo son. Primero veamos la opción `-n`, usada para ordenamiento numérico. Con esta opción, es posible ordenar valores basándonos en valores numéricos. Podemos demostrarlo ordenado los resultados del comando `du` para determinar los usuarios con más espacio en disco. Normalmente, el comando du lista los resultados de un sumario por orden de la ruta:

```
[me@linuxbox ~]$ du -s /usr/share/* | head
252 /usr/share/aclocal
96 /usr/share/acpi-support
8 /usr/share/adduser
196 /usr/share/alacarte
344 /usr/share/alsa
8 /usr/share/alsa-base
12488 /usr/share/anthy
8 /usr/share/apmd
21440 /usr/share/app-install
48 /usr/share/application-registry
```

En este ejemplo, entubamos el resultado en `head` para limitar los resultados a las primeras diez líneas. Podemos producir una lista ordenada numéricamente para mostrar los diez mayores consumidores de espacio de esta forma:

```
[me@linuxbox ~]$ du -s /usr/share/* | sort -nr | head
509940 /usr/share/locale-langpack
242660 /usr/share/doc
197560 /usr/share/fonts
179144 /usr/share/gnome
146764 /usr/share/myspell
144304 /usr/share/gimp
135880 /usr/share/dict
76508 /usr/share/icons
68072 /usr/share/apps
62844 /usr/share/foomatic
```

Usando las opciones `-nr`, producimos un orden numérico inverso, con los mayores valores apareciendo primero en los resultados. Este orden funciona porque los valores numéricos están al principio de cada línea. Pero, ¿qué pasa si queremos ordenar una lista basándonos en algún valor localizado dentro de la línea? Por ejemplo, los resultados de un `ls -l`:

```
[me@linuxbox ~]$ ls -l /usr/bin | head
total 152948
-rwxr-xr-x 1 root root 34824 2008-04-04 02:42 [
-rwxr-xr-x 1 root root 101556 2007-11-27 06:08 a2p
-rwxr-xr-x 1 root root 13036 2008-02-27 08:22 aconnect
-rwxr-xr-x 1 root root 10552 2007-08-15 10:34 acpi
-rwxr-xr-x 1 root root 3800 2008-04-14 03:51 acpi_fakekey
-rwxr-xr-x 1 root root 7536 2008-04-19 00:19 acpi_listen
-rwxr-xr-x 1 root root 3576 2008-04-29 07:57 addpart
-rwxr-xr-x 1 root root 20808 2008-01-03 18:02 addr2line
-rwxr-xr-x 1 root root 489704 2008-10-09 17:02 adept_batch
```

Ignorando, por el momento, que `ls` puede ordenar sus resultados por tamaño, podríamos usar `sort` par ordenar la lista por tamaño de archivo, así:

```
[me@linuxbox ~]$ ls -l /usr/bin | sort -nr -k 5 | head
-rwxr-xr-x 1 root root 8234216 2008-04-07 17:42 inkscape
-rwxr-xr-x 1 root root 8222692 2008-04-07 17:42 inkview
-rwxr-xr-x 1 root root 3746508 2008-03-07 23:45 gimp-2.4
-rwxr-xr-x 1 root root 3654020 2008-08-26 16:16 quanta
-rwxr-xr-x 1 root root 2928760 2008-09-10 14:31 gdbtui
-rwxr-xr-x 1 root root 2928756 2008-09-10 14:31 gdb
-rwxr-xr-x 1 root root 2602236 2008-10-10 12:56 net
-rwxr-xr-x 1 root root 2304684 2008-10-10 12:56 rpcclient
-rwxr-xr-x 1 root root 2241832 2008-04-04 05:56 aptitude
-rwxr-xr-x 1 root root 2202476 2008-10-10 12:56 smbcacls
```

Muchos usos de `sort` implican el procesado de datos tabulares, como los resultados del comando `ls` anterior. Si aplicamos terminología de bases de datos a la tabla anterior, diríamos que cada fila es un registro consistente en múltiples campos, como los atributos de archivo, contador de enlaces,

nombre de archivo, tamaño de archivo y así sucesivamente. `sort` puede procesar campos individuales. En términos de bases de datos, podemos especificar uno o más *campos clave* para usar como *claves de ordenamiento*. En el ejemplo anterior, especificamos las opciones `n` y `r` para realizar un ordenamiento numérico inverso y especificar `-k 5` para hacer que `sort` use el quinto campo como la clave de ordenamiento.

La opción `k` es muy interesante y tiene muchas funciones, pero primero tenemos que hablar sobre como `sort` define los campos. Consideremos un archivo de texto muy simple consistente en una única línea que contenga el nombre del autor:

```
William Shotts
```

Por defecto, `sort` ve esta línea con dos campos. El primer campo contiene los caracteres:

```
"William"
```

y el segundo campo contiene los caracteres:

```
" Shotts"
```

lo que significa que los caracteres de espacio en blanco (espacios y tabuladores) se usan como delimitadores entre los campos y que los delimitadores se incluyen en el campo cuando se realiza el ordenado.

Mirando de nuevo a la línea de la salida de nuestro `ls`, podemos ver que una línea contiene ocho campos y que el quinto campo es el tamaño del archivo:

```
-rwxr-xr-x 1 root root 8234216 2008-04-07 17:42 inkscape
```

Para nuestra próxima serie de experimentos, consideraremos el siguiente archivo que contiene la historia de tres distribuciones Linux populares desarrolladas entre 2006 y 2008. Cada línea del archivo tiene tres campos: el nombre de la distribución, el número de versión y la fecha de publicación en formato MM/DD/YYYY:

```
SUSE 10.2 12/07/2006
Fedora 10 11/25/2008
SUSE 11.0 06/19/2008
Ubuntu 8.04 04/24/2008
Fedora 8 11/08/2007
SUSE 10.3 10/04/2007
Ubuntu 6.10 10/26/2006
Fedora 7 05/31/2007
Ubuntu 7.10 10/18/2007
Ubuntu 7.04 04/19/2007
SUSE 10.1 05/11/2006
Fedora 6 10/24/2006
Fedora 9 05/13/2008
Ubuntu 6.06 06/01/2006
Ubuntu 8.10 10/30/2008
Fedora 5 03/20/2006
```

Usando un editor de texto (quizá `vim`), introduciremos estos datos y nombraremos al archivo

resultante `distros.txt`.

A continuación, trataremos de ordenar el archivo y observaremos el resultado:

```
[me@linuxbox ~]$ sort distros.txt
Fedora 10 11/25/2008
Fedora 5 03/20/2006
Fedora 6 10/24/2006
Fedora 7 05/31/2007
Fedora 8 11/08/2007
Fedora 9 05/13/2008
SUSE 10.1 05/11/2006
SUSE 10.2 12/07/2006
SUSE 10.3 10/04/2007
SUSE 11.0 06/19/2008
Ubuntu 6.06 06/01/2006
Ubuntu 6.10 10/26/2006
Ubuntu 7.04 04/19/2007
Ubuntu 7.10 10/18/2007
Ubuntu 8.04 04/24/2008
Ubuntu 8.10 10/30/2008
```

Bien, casi ha funcionado. El problema aparece en el ordenamiento de los números de versión de Fedora. Como un "1" viene antes que un "5" en el listado de caracteres, la versión "10" aparece arriba mientras que la versión "9" aparece al final.

Para arreglar este problema vamos a tener que ordenar por múltiples claves. Queremos realizar un orden alfabético en el primer campo y luego un orden numérico en el segundo campo. `sort` permite múltiples instancias de la opción `-k` así que pueden especificarse múltiples claves de ordenamiento. De hecho, una clave puede incluir un rango de campos. Si no se especifica un rango (como ha sido el caso de nuestros ejemplos anteriores), `sort` usa una clave que comienza con el campo especificado y se extiende hasta el final de la línea. Aquí tenemos la sintaxis de nuestro ordenamiento multi-clave:

```
[me@linuxbox ~]$ sort --key=1,1 --key=2n distros.txt
Fedora 5 03/20/2006
Fedora 6 10/24/2006
Fedora 7 05/31/2007
Fedora 8 11/08/2007
Fedora 9 05/13/2008
Fedora 10 11/25/2008
SUSE 10.1 05/11/2006
SUSE 10.2 12/07/2006
SUSE 10.3 10/04/2007
SUSE 11.0 06/19/2008
Ubuntu 6.06 06/01/2006
Ubuntu 6.10 10/26/2006
Ubuntu 7.04 04/19/2007
Ubuntu 7.10 10/18/2007
Ubuntu 8.04 04/24/2008
Ubuntu 8.10 10/30/2008
```

Aunque hemos usado la forma larga de la opción para que se vea más claramente, `-k 1, 1 -k 2n` sería completamente equivalente. En la primera instancia de la opción de clave, especificamos un rango de campos a incluir en la primera clave. Como queremos limitar el orden al primer campo solamente, especificamos `1, 1` que significa "comienza en el campo uno y termina en el campo uno." En la segunda instancia, hemos especificado `2n`, que significa que el campo 2 es el campo de ordenado y que el orden debería ser numérico. Puede incluirse una letra de opción al final de un especificador de clave para indicar el tipo de orden a realizar. Estas letras de opción son las mismas que las opciones globales del programa `sort`: `b` (ignora los espacios en blanco delanteros), `n` (orden numérico), `r` (orden inverso), y así sucesivamente.

El tercer campo de nuestra lista contiene una fecha con un formato inapropiado para ordenarlo. En los ordenadores, las fechas están normalmente formateadas como YYYY-MM-DD para hacer más fácil el ordenamiento cronológico, pero la nuestra está en formato Americano MM/DD/YYYY. ¿Cómo podemos ordenar esta lista por orden cronológico?

Afortunadamente, `sort` nos da una forma. La clave de opción permite especificar *compensaciones* dentro de campos, por lo que podemos definir claves dentro de campos:

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt
Fedora 10 11/25/2008
Ubuntu 8.10 10/30/2008
SUSE 11.0 06/19/2008
Fedora 9 05/13/2008
Ubuntu 8.04 04/24/2008
Fedora 8 11/08/2007
Ubuntu 7.10 10/18/2007
SUSE 10.3 10/04/2007
Fedora 7 05/31/2007
Ubuntu 7.04 04/19/2007
SUSE 10.2 12/07/2006
Ubuntu 6.10 10/26/2006
Fedora 6 10/24/2006
Ubuntu 6.06 06/01/2006
SUSE 10.1 05/11/2006
Fedora 5 03/20/2006
```

Especificando `-k 3.7` le decimos a `sort` que use una clave de ordenación que comience en el séptimo carácter del tercer campo, que corresponde al principio del año. De igual forma, especificamos `-k 3.1` y `-k 3.4` para aislar las zonas del día y el mes de la fecha. También añadimos las opciones n y r para realizar un ordenado numérico inverso. La opción b se incluye para suprimir los espacios delanteros (cuyo número varía según la línea, afectando de este modo al resultado del ordenado) en el campo fecha.

Algunos archivos no usan tabuladores ni espacios como delimitadores de campos; por ejemplo, el archivo `/etc/passwd`:

```
[me@linuxbox ~]$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
```

```
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

Los campos en este archivo están delimitados por dos puntos `(:)`, así que ¿cómo ordenamos este archivo usando campos clave? `sort` cuenta con la opción `-t` para definir el carácter separador de campos. Para ordenar el archivo `passwd` por el séptimo campo (el shell por defecto de la cuenta), podríamos hacer ésto:

```
[me@linuxbox ~]$ sort -t ':' -k 7 /etc/passwd | head
me:x:1001:1001:Myself,,,:/home/me:/bin/bash
root:x:0:0:root:/root:/bin/bash
dhcp:x:101:102::/nonexistent:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
klog:x:103:104::/home/klog:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
```

Especificando el carácter de los dos puntos como separador de campos, podemos ordenar por el séptimo campo.

uniq

Comparándolo con `sort`, el programa `uniq` es un peso pluma. `uniq` realiza una tarea que puede parecer trivial. Cuando, le damos un archivo ordenado (incluida la entrada estándar), elimina las líneas duplicadas y manda los resultados a la salida estándar. A menudo se usa junto con `sort` para limpiar la salida de duplicados.

Consejo: Mientras que `uniq` es una herramienta tradicional de Unix a menudo usada con `sort`, la versión GNU de `sort` soporta la opción `-u`, que elimina los duplicados de la salida ordenada.

Hagamos un archivo de texto para probarlo:

```
[me@linuxbox ~]$ cat > foo.txt
a
b
c
a
b
c
```

Recuerda pulsar `Ctrl-d` para finalizar la salida estándar. Ahora, si ejecutamos `uniq` en nuestro archivo de texto:

```
[me@linuxbox ~]$ uniq foo.txt
a
b
```

```
c
a
b
c
```

los resultados no son diferentes de nuestro archivo original; los duplicados no han sido eliminados. Para que `uniq` haga realmente su trabajo, la entrada debe ser ordenada primero:

```
[me@linuxbox ~]$ sort foo.txt | uniq
a
b
c
```

Ésto es porque `uniq` sólo elimina las líneas duplicadas que son adyacentes a otra.

`uniq` tiene varias opciones. Aquí están las más comunes:

Tabla 20-2: Opciones comunes de uniq

Opción	Descripción
`-c`	Muestra una lista de líneas duplicadas precedidas por el número de veces que aparece la línea.
`-d`	Sólo muestra las líneas repetidas, en lugar de las líneas únicas.
`-f n`	Ignora n campos precedentes en cada línea. Los campos se separan por espacios en blanco como en `sort`; sin embargo, al contrario que en `sort`, `uniq` no tiene una opción para configurar un separador de campos alternativo.
`-i`	Ignora las mayúsculas durante la comparación de líneas.
`-s` *n*	Se salta (ignora) los *n* primeros caracteres de cada línea.
`-u`	Sólo muestra líneas únicas. Es la configuración por defecto.

Aquí vemos `uniq` usado para mostrar el número de duplicados encontrados en nuestro archivo de texto, usando la opción `-c`:

```
[me@linuxbox ~]$ sort foo.txt | uniq -c
2 a
2 b
2 c
```

Reordenando

Los tres programas siguientes que vamos a ver se usan para extraer columnas de nuestro archivo de texto y recombinarlas de forma útil.

cut

El programa `cut` se usa para extraer una sección de texto de una línea y pasar la sección extraída a

la salida estándar. Puede aceptar múltiples argumentos de archivo o entradas de la entrada estándar.

Especificar la sección de la línea a extraer es un tanto incómodo y se especifica usando las siguientes opciones:

Tabla 20-3: Opciones de selección de cut

Opción	Descripción
`-c` *`lista_de_caracteres`*	Extrae al porción de línea definida por *lista_de_caracteres*. La lista puede consistir en uno o más rangos numéricos separados por comas.
`-f` *`lista_de_campos`*	Extrae uno o más campos de la línea como esté definido por *lista_de_campos*. La lista puede contener uno o más campos o rangos de campos separados por comas.
`-d` *`carácter_delim`*	Cuando especificamos `-f`, usamos *carácter_delim* como el carácter delimitador de campos. Por defecto, los campos deben ser separados por un carácter de tabulador.
`--complement`	Extrae la línea de texto completa, excepto aquellas porciones especificadas por `-c` y/o `-f`.

Como podemos ver, la forma en que `cut` extrae el texto es más bien inflexible. cut se usa mejor para extraer texto de archivos que son producidos por otros programas, en lugar de texto escrito directamente por humanos. Echaremos un vistazo a nuestro archivo `distros.txt` para ver si está suficientemente "limpio" para ser un buen espécimen para nuestros ejemplos con `cut`. Si usamos `cat` con la opción `-A`, podemos ver si el archivo cumple nuestros requisitos de campos separados por tabuladores:

```
[me@linuxbox ~]$ cat -A distros.txt
SUSE^I10.2^I12/07/2006$
Fedora^I10^I11/25/2008$
SUSE^I11.0^I06/19/2008$
Ubuntu^I8.04^I04/24/2008$
Fedora^I8^I11/08/2007$
SUSE^I10.3^I10/04/2007$
Ubuntu^I6.10^I10/26/2006$
Fedora^I7^I05/31/2007$
Ubuntu^I7.10^I10/18/2007$
Ubuntu^I7.04^I04/19/2007$
SUSE^I10.1^I05/11/2006$
Fedora^I6^I10/24/2006$
Fedora^I9^I05/13/2008$
Ubuntu^I6.06^I06/01/2006$
Ubuntu^I8.10^I10/30/2008$
Fedora^I5^I03/20/2006$
```

Tiene buena pinta. Sin espacios intermedios, sólo caracteres de tabulador entre los campos. Como el archivo usa tabuladores en lugar de espacios, usaremos la opción `-f` para extraer un campo:

```
[me@linuxbox ~]$ cut -f 3 distros.txt
```

```
12/07/2006
11/25/2008
06/19/2008
04/24/2008
11/08/2007
10/04/2007
10/26/2006
05/31/2007
10/18/2007
04/19/2007
05/11/2006
10/24/2006
05/13/2008
06/01/2006
10/30/2008
03/20/2006
```

Como nuestro archivo `distros` está delimitado por tabuladores, es mejor usar `cut` para extraer campos en lugar de caracteres. Ésto es porque cuando un archivo está delimitado por tabuladores, es muy improbable que cada línea contenga el mismo número de caracteres, que hace que calcular posiciones de caracteres dentro de la línea sea complicado o imposible. En nuestro ejemplo anterior, sin embargo, hemos extraído un campo que afortunadamente contiene datos de igual longitud, así que podemos mostrar como funciona la extracción de caracteres extrayendo el año de cada línea:

```
[me@linuxbox ~]$ cut -f 3 distros.txt | cut -c 7-10
2006
2008
2008
2008
2007
2007
2006
2007
2007
2007
2006
2006
2008
2006
2008
2006
```

Ejecutando `cut` una segunda vez en nuestra lista, podemos extraer las posiciones de caracteres de la 7 a la 10, que corresponden al año en nuestro campo fecha. La notación `7-10` es un ejemplo de rango. La man page de `cut` contiene una descripción completa de como pueden especificarse los rangos.

Cuando trabajamos con campos, es posible especificar un delimitador de campo diferente en lugar de el carácter tabulador. Aquí extraeremos el primer campo del archivo `/etc/passwd`:

```
[me@linuxbox ~]$ cut -d ':' -f 1 /etc/passwd | head
root
```

```
daemon
bin
sys
sync
games
man
lp
mail
news
```

Usando la opción -d, podemos especificar el punto como carácter delimitador de campos.

Expandiendo los tabuladores

Nuestro archivo `distros.txt` está perfectamente formateado para extraer campos usando `cut`. Pero ¿qué si queremos un archivo que pueda ser completamente manipulado por `cut` con caracteres, en lugar de con campos? Esto requeriría que reemplazáramos los caracteres tabuladores del archivo por el correspondiente número de espacios. Afortunadamente, el paquete GNU Coreutils incluye una herramienta para ésto. Se llama `expand`, este programa acepta tanto uno como varios argumentos de archivos o entrada estándar, y produce un texto modificado en la salida estándar.

Si procesamos nuestro archivo `distros.txt` con `expand`, podemos usar `cut -c` para extraer cualquier rango de caracteres del archivo. Por ejemplo, podríamos usar el siguiente comando para extraer el año de salida de nuestra lista, expandiendo el archivo y usando `cut` para extraer cada carácter de la vigésimo tercera posición al final de la línea:

```
[me@linuxbox ~]$ expand distros.txt | cut -c 23-
```

Coreutils también aporta el programa `unexpand` para sustituir tabuladores por espacios.

paste

El comando `paste` hace lo contrario de `cut`. En lugar de extraer una columna de texto de un archivo, añade una o más columnas de texto a un archivo. Lo hace leyendo múltiples archivos y combinando los campos encontrados en cada archivo en una única cadena en la entrada estándar. Como `cut`, `paste` acepta múltiples argumentos de archivo y/o entrada estándar. Para demostrar como opera `paste`, realizaremos una cirugía a nuestro archivo `distros.txt` para producir una lista cronológica de lanzamientos.

De nuestro trabajo anterior con `sort`, primero produciremos una lista de distribuciones ordenadas por fecha y guardaremos el resultado en un archivo llamado `distros-by-date.txt`:

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt > dis
tros-by-date.txt
```

A continuación, usaremos `cut` para extraer los dos primeros campos del archivo (el nombre de distribución y la versión), y almacenar el resultado en un archivo llamado `distro-versions.txt`:

```
[me@linuxbox ~]$ cut -f 1,2 distros-by-date.txt > distros-versions.t
xt
[me@linuxbox ~]$ head distros-versions.txt
Fedora 10
Ubuntu 8.10
SUSE 11.0
Fedora 9
Ubuntu 8.04
Fedora 8
Ubuntu 7.10
SUSE 10.3
Fedora 7
Ubuntu 7.04
```

La parte final de la preparación es extraer la fecha de liberación y almacenarla en un archivo llamado `distro-dates.txt`:

```
[me@linuxbox ~]$ cut -f 3 distros-by-date.txt > distros-dates.txt
[me@linuxbox ~]$ head distros-dates.txt
11/25/2008
10/30/2008
06/19/2008
05/13/2008
04/24/2008
11/08/2007
10/18/2007
10/04/2007
05/31/2007
04/19/2007
```

Ahora tenemos las partes que necesitamos. Para completar el proceso, usamos `paste` para colocar la columna de fechas delante de las de el nombre de la distribución y las versiones, de forma que creando así una lista cronológica. Ésto se hace simplemente usando `paste` y ordenando sus argumentos en el orden deseado:

```
[me@linuxbox ~]$ paste distros-dates.txt distros-versions.txt
11/25/2008 Fedora 10
10/30/2008 Ubuntu 8.10
06/19/2008 SUSE 11.0
05/13/2008 Fedora 9
04/24/2008 Ubuntu 8.04
11/08/2007 Fedora 8
10/18/2007 Ubuntu 7.10
10/04/2007 SUSE 10.3
05/31/2007 Fedora 7
04/19/2007 Ubuntu 7.04
12/07/2006 SUSE 10.2
10/26/2006 Ubuntu 6.10
10/24/2006 Fedora 6
06/01/2006 Ubuntu 6.06
05/11/2006 SUSE 10.1
```

```
03/20/2006 Fedora 5
```

join

En algunos aspectos, `join` es como `paste` en cuanto a que añade columnas a un archivo, pero usa una forma única para hacerlo. Un *join* es una operación normalmente asociada con las *bases de datos relacionales* donde los datos de múltiples *tablas* con un campo clave compartido se combinan para dar el resultado deseado. El programa `join` realiza la misma operación. Une datos de múltiples archivos basándose en un campo clave compartido.

Para ver como se usa una operación join en una base de datos relacional, imaginemos una base de datos muy pequeña consistente en dos tablas, cada una con un único registro. La primera tabla, llamada CUSTOMERS, tiene tres campos: un número de cliente (CUSTNUM), el nombre del cliente (FNAME) y el apellido del cliente (LNAME):

CUSTNUM	FNAME	LNAME
========	=====	======
4681934	John	Smith

La segunda tabla se llama ORDERS y contiene cuatro campos: un número de pedido (ORDERNUM), el número de cliente (CUSTNUM), la cantidad (QUAN) y el producto pedido (ITEM).

ORDERNUM	CUSTNUM	QUAN	ITEM
========	=======	====	====
3014953305	4681934	1	Blue Widget

Fíjate que ambas tablas comparten el campo CUSTNUM. Ésto es importante, ya que permite la relación entre las tablas.

Realizar una operación join nos permitiría combinar los campos en las dos tablas para conseguir un resultado útil, como preparar un pedido. Usando los valores coincidentes en los campos CUSTNUM de ambas tablas, una operación join podría producir lo siguiente:

FNAME	LNAME	QUAN	ITEM
=====	=====	====	====
John	Smith	1	Blue Widget

Para probar el programa `join`, necesitaremos crear un par de archivos con una clave compartida. Para hacerlo, usaremos nuestro archivo `distros-by-date.txt`. Desde este archivo, construiremos dos archivos adicionales, uno conteniendo las fechas de lanzamiento (que será nuestra clave compartida para esta demostración) y los nombres de la versión:

```
[me@linuxbox ~]$ cut -f 1,1 distros-by-date.txt > distros-
names.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-names.txt >
distroskey-names.txt
[me@linuxbox ~]$ head distros-key-names.txt
11/25/2008 Fedora
10/30/2008 Ubuntu
06/19/2008 SUSE
05/13/2008 Fedora
04/24/2008 Ubuntu
```

```
11/08/2007 Fedora
10/18/2007 Ubuntu
10/04/2007 SUSE
05/31/2007 Fedora
04/19/2007 Ubuntu
```

y el segundo archivo, que contiene las fechas de lanzamiento y los números de versión:

```
[me@linuxbox ~]$ cut -f 2,2 distros-by-date.txt > distros-
vernums.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-vernums.txt >
distro s-key-vernums.txt
[me@linuxbox ~]$ head distros-key-vernums.txt
11/25/2008 10
10/30/2008 8.10
06/19/2008 11.0
05/13/2008 9
04/24/2008 8.04
11/08/2007 8
10/18/2007 7.10
10/04/2007 10.3
05/31/2007 7
04/19/2007 7.04
```

Ahora tenemos dos archivos con una clave compartida (el campo "fecha de lanzamiento"). Es importante señalar que los archivos deben estar ordenados por el campo clave para que `join` funcione apropiadamente.

```
[me@linuxbox ~]$ join distros-key-names.txt distros-key-
vernums.txt |
head
11/25/2008 Fedora 10
10/30/2008 Ubuntu 8.10
06/19/2008 SUSE 11.0
05/13/2008 Fedora 9
04/24/2008 Ubuntu 8.04
11/08/2007 Fedora 8
10/18/2007 Ubuntu 7.10
10/04/2007 SUSE 10.3
05/31/2007 Fedora 7
04/19/2007 Ubuntu 7.04
```

Fíjate también que, `join` usa el espacio en blanco como el delimitador de campos de entrada y un espacio individual como el el delimitador de campos de salida. Este comportamiento puede modificarse especificando opciones. Mira la man page de `join` para más detalles.

Comparando texto

A menudo es útil comparar versiones de archivos de texto. Para administradores de sistemas y desarrolladores de software, es particularmente importante. Un administrador de sistemas podría, por ejemplo, necesitar comparar un archivo de configuración existente con una versión anterior para diagnosticar un problema del sistema. Igualmente, un programador necesita frecuentemente ver que

cambios se le han realizado a los programas a lo largo del tiempo.

comm

El programa `comm` compara dos archivos de texto y muestra las líneas que son únicas y las líneas que tienen iguales. Para demostrarlo, crearemos dos archivos de texto casi idénticos usando `cat`:

```
[me@linuxbox ~]$ cat > file1.txt
a
b
c
d [
me@linuxbox ~]$ cat > file2.txt
b
c
d
e
```

A continuación, compararemos los dos archivos usando `comm`:

```
[me@linuxbox ~]$ comm file1.txt file2.txt
a
b
c
d
e
```

Como podemos ver, `comm` produce tres columnas de salida. La primera columna contiene líneas únicas en el primer archivo; la segunda columna, las líneas únicas en el segundo archivo; la tercera columna contiene las líneas compartidas por ambos archivos. `comm` soporta opciones en la forma `-n` donde n es 1, 2 o 3. Cuando se usan, estas opciones especifican que columna(s) suprimir. Por ejemplo, si sólo queremos obtener las líneas compartidas por ambos archivos, suprimiremos la salida de las columnas uno y dos:

```
[me@linuxbox ~]$ comm -12 file1.txt file2.txt
b
c

d
```

diff

Como el programa `comm`, `diff` se usa para detectar diferencias entre archivos. Sin embargo, `diff` es una herramienta mucho más compleja, soporta muchos formatos de salida y la capacidad de procesar grandes colecciones de archivos de texto a la vez. diff a menudo se usa por los desarrolladores de software para examinar cambios entre versiones diferentes de código fuente de programa, y además tiene la capacidad de examinar recursivamente directorios de código fuente, a menudo llamados *árboles fuente*. Un uso común para `diff` es crear *archivos diff* o *parches* que se usan por programas como `patch` (que veremos pronto) para convertir una versión de un archivo (o archivos) a otra versión.

Si usamos `diff` para ver nuestros archivos de ejemplo previos:

```
[me@linuxbox ~]$ diff file1.txt file2.txt
1d0
< a
4a4
> e
```

vemos su estilo de salida por defecto: una breve descripción de las diferencias entre dos archivos. En el formato por defecto, cada grupo de cambios va precedido por un *comando de cambio* en la forma de un *rango operación rango* para describir las posiciones y tipos de cambios requeridos para convertir el primer archivo en el segundo:

Tabla 20-4: Comandos de cambio de diff

Cambio	Descripción
`r1ar2`	Añade las líneas en la posición *r2* del segundo archivo a la posición *r1* del primer archivo.
`r1cr2`	Cambia (reemplaza) las lineas en la posición *r1* con las líneas en la posición *r2* en el segundo archivo.
`r1dr2`	Borra las líneas en el primer archivo en la posición *r1*, que habrían aparecido en el rango *r2* del segundo archivo.

En este formato, un rango es una lista separada por comas de la línea inicial y la línea final. Aunque este formato es el que viene por defecto (principalmente por conformidad con POSIX y retrocompatibilidad con versiones tradicionales de Unix de diff), no está tan ampliamente usado como otros formatos opcionales. Dos de los formatos más populares son el *formato contextual* y el *formato unificado*.

Cuando lo vemos usando el formato contextual (la opción `-c`), veremos ésto:

```
[me@linuxbox ~]$ diff -c file1.txt file2.txt
*** file1.txt 2008-12-23 06:40:13.000000000 -0500
--- file2.txt 2008-12-23 06:40:34.000000000 -0500
***************
*** 1,4 ****
- a
b
c
d
--- 1,4 ----
b
c
```

```
d
+ e
```

La salida comienza con los nombres de los dos archivos y sus fechas de modificación. El primer archivo está marcado con asteriscos y el segundo archivo está marcado con guiones. A lo largo del resto de la lista, estos marcadores simbolizarán sus respectivos archivos. A continuación, vemos grupos de cambios, incluyendo el número de líneas de contexto circundantes por defecto. En el primer grupo, vemos:

```
*** 1,4 ***
```

que indica lineas de la 1 a la 4 en el primer archivo. A continuación vemos:

```
--- 1,4 ---
```

que indica lineas de la 1 a la 4 en el segundo archivo. Dentro de un grupo de cambios, las líneas comienzan con uno de estos cuatro indicadores:

Tabla 20-5: Indicadores de cambios de diff en formato contextual

Indicador	Significado
`blank`	Un línea mostrada de contexto. No indica ninguna diferencia entre los dos archivos.
`-`	Un línea borrada. Esta línea aparecerá en el primer archivo pero no en el segundo.
`+`	Una línea añadida. Esta línea aparecerá en el segundo archivo pero no en el primero.
`!`	Una línea modificada. Las dos versiones de la línea se mostrarán, cada una en su respectiva sección del grupo de cambio.

El formato unificado es similar al formato contextual pero es más conciso. Se especifica con la opción `-u`:

```
[me@linuxbox ~]$ diff -u file1.txt file2.txt
--- file1.txt 2008-12-23 06:40:13.000000000 -0500
+++ file2.txt 2008-12-23 06:40:34.000000000 -0500
@@ -1,4 +1,4 @@
-a
b
c
d
+e
```

La diferencia más destacable entre los formatos contextual y unificado es la eliminación de las líneas duplicadas de contexto, haciendo que los resultados del formato unificado sean más cortos que los del formato contextual. En nuestro ejemplo anterior, vemos las marcas de tiempo de los archivos como los del formato contextual, seguidas de la cadena `@@ -1,4 +1,4 @@`. Ésto indica las líneas del primer archivo y las líneas del segundo archivo descritas en el grupo de cambio. Tras ésto están las líneas en sí, con las tres líneas de contexto por defecto. Cada línea comienza con uno de tres posibles caracteres:

Tabla 20-6: *Indicadores de cambios de diff en formato unificado*

Carácter	Significado
`blank`	Esta línea la comparten ambos archivos.
`-`	Esta línea se eliminó del primer archivo.
`+`	Esta línea se añadió al primer archivo

patch

El programa `patch` se usa para aplicar cambios a archivos de texto. Acepta salida de `diff` y generalmente se usa para convertir versiones antiguas de archivos en versiones más nuevas. Consideremos un ejemplo famoso. El kernel Linux es desarrollado por un equipo grande y poco organizado de contribuidores que envían una constante cadena de pequeños cambios al código fuente. El kernel Linux consta de varios millones de líneas de código, mientras que los cambios que hace un contribuidor cada vez son muy pequeños. No tiene sentido que un contribuidor envíe en cada desarrollo un código fuente completo del kernel cada vez que realiza un pequeño cambio. En su lugar, envía un archivo diff. El archivo diff contiene el cambio desde la versión anterior del kernel a la nueva versión con los cambios del contribuidor. El que lo recibe usa el programa `patch` para aplicar el cambio a su propio código fuente. Usar `diff/patch` proporciona dos ventajas significativas:

1. El archivo diff es muy pequeño, comparado con el tamaño completo del código fuente.
2. El archivo diff muestra de forma concisa el cambio a realizar, permitiendo a los revisores del parche evaluarlo rápidamente.

Claro que, `diff/patch` funcionará con cualquier archivo de texto, no sólo con código fuente. Será igualmente aplicable a archivos de configuración o cualquier otro texto.

Para preparar un archivo diff para usarlo con `patch`, la documentación GNU (ver Para Saber Más a continuación) sugiere usar `diff` de la siguiente manera:

```
diff -Naur old_file new_file > diff_file
```

Donde *old_file* y *new_file* son archivos individuales o directorios que contengan archivos. La opción `r` soporta recursividad en el árbol de directorios.

Una vez que el archivo diff ha sido creado, podemos aplicarlo para parchear el archivo antiguo y convertirlo en el nuevo archivo:

```
patch < diff_file
```

Lo demostraremos con nuestro archivo de pruebas:

```
[me@linuxbox ~]$ diff -Naur file1.txt file2.txt > patchfile.txt
[me@linuxbox ~]$ patch < patchfile.txt
patching file file1.txt
[me@linuxbox ~]$ cat file1.txt
b
c
d
e
```

En este ejemplo, hemos creado un archivo diff llamado `patchfile.txt` y luego hemos usado el programa `patch` para aplicar el parche. Fíjate que no hemos tenido que especificar un archivo de destino para `patch`, ya que el archivo diff (en formato unificado) ya contiene los nombres de archivo en la cabecera. Una vez que se aplica el parche, podemos ver que `file1.txt` ahora coincide con `file2.txt`.

`patch` tiene una gran número de opciones, y hay programas adicionales que pueden usarse para analizar y editar parches.

Editando sobre la marcha

Nuestra experiencia con editores de texto ha sido muy *interactiva*, ya que movemos el cursor manualmente, y luego escribimos nuestros cambios. Sin embargo, hay formas *no-interactivas* de editar texto también. Es posible, por ejemplo, aplicar una serie de cambios a múltiples archivos con un único comando.

tr

El programa `tr` se usa para *transliterar* caracteres. Podemos pensar que esto es un tipo de operación de reemplazo y búsqueda basada en caracteres. La transliteración es el proceso de cambiar caracteres de un alfabeto a otro. Por ejemplo, convertir caracteres de minúscula a mayúscula es transliteración. Podemos realizar esa conversión con `tr` como sigue:

```
[me@linuxbox ~]$ echo "lowercase letters" | tr a-z A-Z
LOWERCASE LETTERS
```

Como podemos ver, `tr` funciona con la entrada estándar, y muestra sus resultados en la salida estándar. `tr` acepta dos argumentos: una serie de caracteres de donde convertir y una serie correspondiente de caracteres hacia donde convertir. Las series de caracteres pueden expresarse de tres formas diferentes:

1. Una lista enumerada. Por ejemplo, `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
2. Un rango de caracteres. Por ejemplo, `A-Z`. Fíjate que este método está sujeto a menudo a los mismos problemas que otros comandos, debido al tipo de orden local, y por lo tanto hay que usarlo con precaución.
3. Clases de caracteres POSIX. Por ejemplo, `[:upper;]`.

En la mayoría de los casos, ambas series de caracteres serán de igual longitud; sin embargo, es posible que la primera serie sea más grande que la segunda, particularmente y queremos convertir muchos caracteres a uno único:

```
[me@linuxbox ~]$ echo "lowercase letters" | tr [:lower:] A
AAAAAAAAA AAAAAAA
```

Además de la transliteración, `tr` permite que los caracteres sean simplemente eliminados de la cadena de entrada. Antes, en este capítulo, vimos el problema de convertir archivos de texto MS-DOS a texto estilo-Unix. Para realizar esta conversión, los caracteres de retorno de carro necesitan ser eliminados del final de cada línea. Ésto puede realizarse con `tr` de la forma siguiente:

```
tr -d '\r' < dos_file > unix_file
```

donde *dos_file* es el archivo a convertir y *unix_file* es el resultado. Esta forma del comando utiliza la

secuencia de escape `\r` para representar el carácter retorno de carro. Para ver una lista completa de las secuencias y clases de caracteres que soporta `tr`, prueba:

```
[me@linuxbox ~]$ tr --help
```

`tr` puede realizar otro truco también. Usando la opción `-s`, `tr` puede "squeeze" (borrar) las instancias repetidas de un caracter:

```
[me@linuxbox ~]$ echo "aaabbbccc" | tr -s ab
abccc
```

Aquí tenemos un cadena que contiene caracteres repetidos. Especificando la lista "ab" a `tr`, eliminamos las instancias repetidas de las letras de la lista, mientras que deja los caractres que no aparecen en la lista ("c") sin cambios. Fíjate que los caracteres repetidos debe estar contiguos. Si no lo están:

```
[me@linuxbox ~]$ echo "abcabcabc" | tr -s ab
abcabcabc
```

la eliminación no tendrá efecto.

sed

El nombre `sed` es una abreviatura de *stream editor* (editor de cadenas). Realiza edición de texto en una cadena de texto, en una serie de archivos especificados o en la entrada estándar. `sed` es un programa potente y algo complejo (hay libros enteros sobre él), así que no lo veremos completamente aquí.

En general, la forma en que funciona `sed` es que se le da un comando simple de edición (en la línea de comandos) o el nombre de un archivo de scripts que contenga múltiples comandos, y ejecuta estos comandos línea a línea de la cadena de texto. Aquí tenemos un ejemplo muy simple de `sed` en acción:

```
[me@linuxbox ~]$ echo "front" | sed 's/front/back/'
back
```

En este ejemplo, producimos una cadena de texto de una palabra usando echo y entubándolo en sed. sed, a continuación, ejecuta la instrucción `s/front/back/` del texto de la cadena y produce la salida "back" como resultado. También podemos considerar este comando de forma parecida al comando "sustitución" (buscar y reemplazar) de `vi`.

Los comandos en `sed` comienzan con una simple letra. En el ejemplo anterior, el comando sustitución se representa con la letra `s` y es seguido por las cadenas a buscar y reemplazar, separadas por el carácter de la barra inclinada como separador. La elección del carácter separador es arbitraria. Por consenso, el carácter de la barra inclinada se usa a menudo, pero `sed` aceptará cualquier carácter que siga inmediatamente al comando como separador. Podríamos realizar el mismo comando de la siguiente forma:

```
[me@linuxbox ~]$ echo "front" | sed 's_front_back_'
back
```

Usar el carácter guion bajo inmediatamente tras el comando, lo convierte en el separador. La capacidad de establecer el separador puede usarse para realizar los comandos más legibles, como veremos.

La mayoría de los comandos en `sed` irán precedidos por una *dirección* , que especifica que línea(s) de la cadena de entrada será editada. Si se omite la dirección, el comando editor se ejecutará en cada línea de la cadena de entrada. La forma más simple de dirección es un número de línea. Podemos añadir uno a nuestro ejemplo:

```
[me@linuxbox ~]$ echo "front" | sed '1s/front/back/'
back
```

Añadiendo la dirección `1` a nuestro comando hacemos que nuestra sustitución se realice en la primera linea de nuestra entrada de una línea. Si especificamos otro número:

```
[me@linuxbox ~]$ echo "front" | sed '2s/front/back/'
front
```

vemos que la edición no se ha realizado, ya que nuestra cadena de entrada no tiene línea 2.

Las direcciones puede ser expresadas de muchas formas. Aquí tenemos las más comunes:

Tabla 20-7: Notación de direcciones en sed

Dirección	Descripción
n	Un número de línea donde *n* es un entero positivo.
$	La última línea
/ *regexp/*	Lineas que coinciden con una expresión regular básica POSIX. Fíjate que la expresión regular se delimita por barras inclinadas. Opcionalmente, la expresión regular puede delimitarse por un carácter alternativo, especificando la expresión con *cregexpc*, donde *c* es el carácter alternativo.
addr1, addr2	Un rango de líneas desde *addr1* a *addr2*, ambos incluidos. Las direcciones pueden estar en cualquiera de las formas que vimos antes.
first~step	Encuentra la línea representada por el número *first*, y luego cada línea siguiente con el intervalo *step*. Por ejemplo 1~2 se refiere a cada línea impar, 5~5 se refiere a la quinta línea y cada quinta línea después de ella.
addr1, +n	Encuentra *addr1* y las siguientes *n* líneas.
addr!	Encuentra todas las líneas excepto *addr*, que puede estar en cualquiera de las formas anteriores.

Probaremos diferentes tipos de direcciones usando el archivo `distros.txt` que usamos antes en este capítulo. Primero, una rango de números de línea:

```
[me@linuxbox ~]$ sed -n '1,5p' distros.txt
SUSE   10.2 12/07/2006
Fedora 10   11/25/2008
```

```
SUSE   11.0 06/19/2008
Ubuntu 8.04 04/24/2008
Fedora 8    11/08/2007
```

En este ejemplo, obtenemos un rango de líneas, comenzando con la línea 1 y continuando hasta la línea 5. Para hacerlo, usamos el comando `p`, que simplemente hace que una línea encontrada se muestre. Para que esto sea efectivo sin embargo, tenemos que incluir la opción `-n` (la opción no auto-imprmir) para hacer que `sed` no muestre cada línea por defecto.

A continuación, probemos algunas expresiones regulares:

```
[me@linuxbox ~]$ sed -n '/SUSE/p' distros.txt
SUSE 10.2 12/07/2006
SUSE 11.0 06/19/2008
SUSE 10.3 10/04/2007
SUSE 10.1 05/11/2006
```

Incluyendo la expresión regular delimitada por barras inclinadas `/SUSE/`, podemos aislar las líneas que la contengan casi de la misma forma que con `grep`.

Finalmente, probaremos la negación añadiendo un signo de exclamación (`!`) a la dirección:

```
[me@linuxbox ~]$ sed -n '/SUSE/!p' distros.txt
Fedora 10   11/25/2008
Ubuntu 8.04 04/24/2008
Fedora 8    11/08/2007
Ubuntu 6.10 10/26/2006
Fedora 7    05/31/2007
Ubuntu 7.10 10/18/2007
Ubuntu 7.04 04/19/2007
Fedora 6    10/24/2006
Fedora 9    05/13/2008
Ubuntu 6.06 06/01/2006
Ubuntu 8.10 10/30/2008
Fedora 5    03/20/2006
```

Aquí vemos el resultado esperado: todas las líneas en el archivo excepto las que coinciden con la expresión regular.

Hasta ahora, hemos visto dos de los comandos de edición de `sed`, `s` y `p`. Aquí tenemos una lista completa de los comandos de edición básicos:

Tabla 20-8: comandos de edición básicos de sed

Comando	Descripción
`=`	Muestra el número de línea actual.
`a`	Añade texto tras la línea actual.
`d`	Borra la línea actual.
`i`	Inserta texto al principio de la línea actual.

`p`	Muestra la línea actual. Por defecto, `sed` imprime cada línea y sólo edita líneas que coinciden con una dirección especificada dentro del archivo. El comportamiento por defecto puede sobrescribirse especificando la opción `-n`.
`q`	Sale de `sed` sin procesar más líneas. Si la opción `-n` no se especifica, muestra la línea actual.
`Q`	Sale de `sed` si procesar más líneas.
`s/regexp/replacement/`	Sustituye el contenido de *replacement* donde se encuentre *regexp*, *replacement* puede incluir el carácter `&`, que es equivalente al texto encontrado por *regexp*. Además, *replacement* puede incluir las secuencias `\1` a `\9`, que son el contenido de las correspondientes subexpresiones de *regexp*. Para saber más sobre ésto, mira el tema de retroreferencias que está más adelante. Tras la barra invertida delantera que sigue a *replacement*, puede especificarse un parámetro para modificar el comportamiento del comando `s`.
`y/set1/set2`	Realiza transliteración convirtiendo caracteres de *set1* a los correspondientes caracteres en *set2*. Fíjate que al contrario que `tr`, `sed` requiere que ambas series tengan la misma longitud.

El comando `s` es de lejos el comando de edición más comúnmente usado. Probaremos sólo algunas de sus capacidades realizando edición en nuestro archivo `distros.txt`. Veremos antes como el campo fecha en `distros.txt` no estaba en formáto "computer-friendly". Ya que la fecha está formateada MM/DD/YYY, y sería mejor (para ordenarla más fácilmente) si el formato fuera YYYY-MM-DD. Realizar este cambio en el archivo a mano consumiría mucho tiempo y sería propenso a errores, pero con `sed`, este cambio puede realizarse en un paso:

```
[me@linuxbox ~]$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\
{4\}\)$/\3-\1-\2/' distros.txt
SUSE   10.2 2006-12-07
Fedora 10   2008-11-25
SUSE   11.0 2008-06-19
Ubuntu 8.04 2008-04-24
Fedora 8    2007-11-08
SUSE   10.3 2007-10-04
Ubuntu 6.10 2006-10-26
Fedora 7    2007-05-31
Ubuntu 7.10 2007-10-18
Ubuntu 7.04 2007-04-19
SUSE   10.1 2006-05-11
Fedora 6    2006-10-24
Fedora 9    2008-05-13
Ubuntu 6.06 2006-06-01
Ubuntu 8.10 2008-10-30
Fedora 5    2006-03-20
```

¡Guau! Es un comando muy feo. Pero funciona. En sólo un paso, hemos cambiado el formato de fecha de nuestro archivo. También es un ejemplo perfecto de por qué las expresiones regulares son llamadas a veces en broma medios "sólo-escritura". Podemos escribirlas, pero a menudo no podemos leerlas. Antes de que estemos tentados de huir de pánico de este comando, veamos como ha sido construido. Primero, sabemos que el comando tendrá está estructura básica:

```
sed 's/regexp/replacement/' distros.txt
```

Nuestro próximo paso es encontrar una expresión regular que aisle la fecha. Como está en formato MM/DD/YYYY y aparece al final de la línea, podemos usar una expresión como esta:

```
[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

que encuentra dos dígitos, una barra, dos dígitos, una barra, cuatro dígitos y el final de la línea. Así que esto se ocupa de *regexp*, pero ¿qué pasa con *replacement*? Para manejar esto, tenemos que introducir una nueva capacidad de las expresiones regulares que aparecen en algunas aplicaciones que usan BRE. Esta capacidad se llama *retroreferencias* y funcionan así: Si aparece la secuencia *\n* en *replacement* donde *n* es un número del 1 al 9, la secuencia se referirá a la subexpresión correspondiente en la expresión regular precedente. Para crear las subexpresiones, simplemente la metemos entre paréntesis así:

```
([0-9]{2})/([0-9]{2})/([0-9]{4})$
```

Ahora tenemos tres subexpresiones. La primera contiene el mes, la segunda contiene el día del mes y la tercera contiene el año. Ahora podemos construir *replacement* como sigue:

```
\3-\1-\2
```

que nos da el año, una barra, el mes, una barra y el día.

Ahora nuestro comando tiene esta pinta:

```
sed 's/([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2/' distros.txt
```

Tenemos dos problemas todavía. El primero es que la barra extra en nuestra expresión confundirá a `sed` cuando trate de interpretar el comando `s`. El segundo es que como `sed`, por defecto, acepta sólo expresiones regulares básicas, varios caracteres en nuestra expresión regular serán tratados como literales, en lugar de como metacaracteres. Podemos resolver ambos problemas con una generosa utilización de barras para escapar los caracteres ofensivos:

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/'
distros.txt
```

¡Y aquí lo tenemos!

Otra característica del comando `s` es el uso de etiquetas opcionales que puede seguir a la cadena de reemplazo. La más importante es la etiqueta `g`, que ordena a `sed` que aplique una búsqueda y reemplazo global a una línea, no sólo a la primera instancia, que es la opción por defecto. Aquí tenemos un ejemplo:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/'
aaaBbbccc
```

Vemos que se ha realizado el reemplazo, pero sólo en la primera instancia de la letra "b", mientras que las siguientes instancias se han dejado sin cambios. Añadiendo la etiqueta `g`, podemos cambiar todas las instancias:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/g'
aaaBBBccc
```

Hasta ahora, sólo le hemos dado a `sed` comandos individuales vía la línea de comandos. También es posible construir comandos más complejos en un archivo de scripts usando la opción `-f`. Para hacer una demostración, usaremos `sed` con nuestro archivo `distros.txt` para construir un informe. Nuestro informe mostrará un título arriba, nuestras fechas modificadas y todos los nombres de las distribuciones convertidos a mayúsculas. Para hacerlo, necesitaremos escribir un script, así que arrancaremos nuestro editor de texto y escribiremos los siguiente:

```
# sed script to produce Linux distributions report
1 i\
\
Linux Distributions Report\
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Guardaremos nuestro script de `sed` como `distros.sed` y lo ejecutaremos así:

```
[me@linuxbox ~]$ sed -f distros.sed distros.txt
Linux Distributions Report
SUSE   10.2 2006-12-07
FEDORA 10   2008-11-25
SUSE   11.0 2008-06-19
UBUNTU 8.04 2008-04-24
FEDORA 8    2007-11-08
SUSE   10.3 2007-10-04
UBUNTU 6.10 2006-10-26
FEDORA 7    2007-05-31
UBUNTU 7.10 2007-10-18
UBUNTU 7.04 2007-04-19
SUSE   10.1 2006-05-11
FEDORA 6    2006-10-24
FEDORA 9    2008-05-13
UBUNTU 6.06 2006-06-01
UBUNTU 8.10 2008-10-30
FEDORA 5    2006-03-20
```

Como podemos ver, nuestro script produce el resultado esperado, pero ¿cómo lo ha hecho? Echemos otro vistazo a nuestro script. Usaremos `cat` para numerar las líneas:

```
[me@linuxbox ~]$ cat -n distros.sed
1 # sed script to produce Linux distributions report
2
3 1 i\
4 \
5 Linux Distributions Report\
6
7 s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)
$/\3-\1-\2/
8 y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

La línea uno de nuestro script es un *comentario*. Como muchos archivos de configuración y

lenguajes de programación en sistemas Linux, los comentarios comienzan con el caracter # y siguen con texto legible por humanos. Los comentarios pueden colocarse en cualquier parte del script (aunque no dentro de los propios comandos) y son útiles para las personas que necesiten identificar y/o mantener el script.

La línea 2 es una línea en blanco. Como los comentarios, las líneas en blanco deben añadirse para mejorar la legibilidad.

Muchos comandos `sed` soportan direcciones de líneas. Aquí se usan para especificar en qué líneas de entrada hay que actuar. Las direcciones de líneas pueden expresarse como números de línea individuales, rangos de números de líneas y el número especial de línea "$" que indica la última línea de la entrada.

Las líneas 3 a 6 contienen texto a insertar en la dirección 1, la primera línea de la entrada. Al comando `i` le sigue la secuencia barra invertida-retorno de carro para producir un retorno de carro escapado, o lo que se llama un *carácter de continuación de línea*. Esta secuencia, que puede usarse en muchas circunstancias incluyendo scripts de shell, permite insertar un retorno de carro en una cadena de texto sin señalar al interprete (en este caso `sed`) que se ha alcanzado el final de la línea. Los comandos `i`, y de igual forma, `a` (que añade texto, en lugar de insertarlo) y `c` (que reemplaza texto), permiten múltiples líneas de texto siempre que cada línea, excepto la última, termine con el carácter continuación de línea. La sexta línea de nuestro script es realmente el final de nuestro texto insertado y termina con un retorno de carro plano en lugar de con un carácter de continuación de línea, señalándole el final al comando `i`.

Nota: Un carácter de continuación de línea lo forman una barra seguida *inmediatamente* por un retorno de carro. No se permiten espacios intermedios.

La línea 7 es nuestro comando de buscar y reemplazar. Como no está precedido por ninguna dirección, cada línea de la cadena de entrada está sometida a su acción.

La línea 8 realiza transliteración de las letras minúsculas a letras mayúsculas. Fíjate que al contrario que `tr`, el comando `y` de sed no soporta rangos de caracteres (por ejemplo, `[a-z]`), ni soporta clases de caracters POSIX. De nuevo, como el comando `y` no está precedido por ninguna dirección, se aplica a cada línea de la cadena de entrada.

ROT13: El anillo decodificador no tan secreto

Un uso divertido de `tr` es realizar un *cifrado de texto* ROT13. ROT13 es un tipo de cifrado trivial basado en un cifrado de sustitución simple. Llamar a ROT13 "cifrado" es ser generosos; "ofuscación de texto" es más correcto. Se usa algunas veces en texto para esconder texto potencialmente ofensivo. El método simplemente mueve cada carácter 13 posiciones hacia delante en el alfabeto. Como esto supone la mitad de los 26 caracteres posibles, aplicando el algoritmo una segunda vez al texto lo restauramos a su forma original. Para realizar este cifrado con `tr`:

```
echo "secret text" | tr a-zA-Z n-za-mN-ZA-M
frperg grkg
```

Realizando el mismo procedimiento una segunda vez obtenemos la traducción:

```
echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M
secret text
```

Numerosos programas de correo electrónico y lectores de noticias de Usenet soportan cifrado ROT13. Wikipedia tiene un buen artículo sobre el asunto:

http://en.wikipedia.org/wiki/ROT13

A la gente que le gusta sed también le gusta...

`sed` es un programa muy capaz, permite realizar tareas de edición muy complejas a cadenas de texo. A menudo se usa para tareas de una línea en lugar de para scripts. Muchos usuarios prefieren otras herramientas para tareas más grandes. Los más populares son `awk` y `perl`. Éstos van más allá de meras herramientas como los programas que hemos visto aquí, y se adentran en el reino de los lenguajes de programación completos. `perl`, en particular, se usa a menudo en lugar de los scripts de shell para muchas tareas de administración y mantenimiento de sistemas, así como un medio muy popular en desarrollos web. awk es un poco más especializado. Su fuerza específica es la capacidad de manipular datos tabulares. Se parece a `sed` en que los programas `awk` normalmente procesan archivos de texto línea a línea, usando un esquema similar al concepto de `sed` de dirección seguida de una acción. Aunque tanto `awk` como `perl` que fuera del objetivo de este libro, son muy buenas experiencias para el usuario de la línea de comandos de Linux.

aspell

La última herramienta que veremos es `aspell`, un corrector ortográfico interactivo. El programa `aspell` es el sucesor de un programa anterior llamado `ispell`, y se puede usar, en mayor parte, como un sucesor. Aunque el programa `aspell` se usa mayormente por otros programas que requieren capacidad de corrección ortográfica, también puede usarse muy eficientemente como una herramienta independiente de la línea de comandos. Tiene la capacidad de comprobar inteligentemente varios tipos de archivos de texto, incluyendo documentos HTML, C/C++, programas, mensajes de correo y otros tipos de textos especializados.

Para comprobar ortográficamente un archivo de texto con un texto de prosa simple, puede usarse así:

```
aspell check textfile
```

donde *textfile* es el nombre del archivo a comprobar. Como ejemplo práctico, crearemos un archivo de texto simple llamado `foo.txt` que contengan algunos errores ortográficos intencionados:

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox jimped over the laxy dog.
```

A continuación, comprobamos el archivo usando `aspell`:

```
[me@linuxbox ~]$ aspell check foo.txt
```

Como `aspell` es interactivo en el modo de comprobación, veremos una pantalla como esta:

```
The quick brown fox jimped over the laxy dog.
1) jumped    6) wimped
2) gimped    7) camped
3) comped    8) humped
```

```
4) limped    9) impede
5) pimped    0) umped
i) Ignore    I) Ignore all
r) Replace   R) Replace all
a) Add       l) Add Lower
b) Abort     x) Exit
?
```

En la parte de arriba de la pantalla, vemos nuestro texto con una palabra sospechosa señalada. En el centro, vemos diez sugerencias ortográficas numeradas del cero al nueve, seguida de una lista de otras opciones posibles. Finalmente, muy abajo, vemos un prompt preparado para aceptar nuestra elección.

Si pulsamos la tecla 1, `aspell` reemplaza la palabra incorrecta por la palabra "jumped" y se mueve a la siguiente palabra mal escrita, que es "laxy". Si seleccionamos el reemplazo "lazy", `aspell` la cambia y termina. Una vez que `aspell` ha terminado, podemos examinar nuestro archivo y ver que los errores han sido corregidos:

```
[me@linuxbox ~]$ cat foo.txt
The quick brown fox jumped over the lazy dog.
```

A menos que se le indique en la línea de comandos la opción `--dont-backup`, `aspell` crea una copia de seguridad del archivo con el texto original añadiéndole la extensión `.bak` al nombre del archivo.

Presumiendo de nuestra habilidad editando con `sed`, pondremos nuestro errores ortográficos de nuevo para que podamos reutilizar nuestro archivo.

```
[me@linuxbox ~]$ sed -i 's/lazy/laxy/; s/jumped/jimped/' foo.txt
```

La opción `-i` de `sed` para editar el archivo "in-situ", significa que en lugar de enviar la salida editada a la salida estándar, reescribirá el archivo con los cambios aplicados. También vemos la capacidad de colocar más de un comando de edición en la línea separándolos con un punto y coma.

A continuación, veremos como `aspell` puede manejar diferentes tipos de archivos de texto. Usando un editor de texto como `vim` (los aventureros pueden probar `sed`), añadiremos algunas etiquetas HTML a nuestro archivo:

```
<html>
 <head>
  <title>Mispelled HTML file</title>
 </head>
 <body>
  <p>The quick brown fox jimped over the laxy dog.</p>
 </body>
</html>
```

Ahora, si tratamos de comprobar la ortografía de nuestro archivo modificado, encontramos un problema. Si lo hacemos así:

```
[me@linuxbox ~]$ aspell check foo.txt
```

tendremos ésto:

```
<html>
 <head>
  <title>Mispelled HTML file</title>
 </head>
 <body>
  <p>The quick brown fox jimped over the laxy dog.</p>
 </body>
</html>
1) HTML  4) Hamel
2) ht ml     5) Hamil
3) ht-ml     6) hotel



i) Ignore    I) Ignore all
r) Replace R) Replace all
a) Add       l) Add Lower
b) Abort    x) Exit
?
```

`aspell` verá el contenido de las etiquetas HTML como errores ortográficos. El problema puede solucionarse incluyendo la opción de modo de comprobación `-H` (HTML), como ésto:

```
[me@linuxbox ~]$ aspell -H check foo.txt
```

que tendrá como resultado ésto:

```
<html>
 <head>
  <title>Mispelled HTML file</title>
 </head>
 <body>
  <p>The quick brown fox jimped over the laxy dog.</p>
 </body>
</html>



1) Mi spelled  6) Misapplied
2) Mi-spelled  7) Miscalled
3) Misspelled  8) Respelled
4) Dispelled   9) Misspell
5) Spelled     0) Misled
i) Ignore      I) Ignore all
r) Replace     R) Replace all
a) Add         l) Add Lower
b) Abort       x) Exit
?
```

El HTML se ignora y sólo las porciones no-HTML del archivo se comprueban. En este modo, el contenido de las etiquetas HTML son ignoradas y no se les comprueba la ortografía. Sin embargo, el contenido de las etiquetas `ALT`, que es bueno que se comprueben, sí se chequean en este modo.

Nota: Por defecto, `aspell` ignorará URLs y direcciones de correo electrónico en el texto. Este comportamiento puede ser modificado con opciones de la línea de comandos. También es posible especificar qué etiquetas son comprobadas y cuales son obviadas. Mira la man page de `aspell` para más detalles.

Resumiendo

En este capítulo, hemos vimos algunas de las muchas herramientas de la línea de comandos que operan sobre textos. En el siguiente capítulo, veremos algunas más. Cierto es que, puede no ser inmediatamente obvio como o porqué tendrías que usar estas herramientas en el día a día, por lo que hemos tratado de mostrar algunos ejemplos semi-prácticos de su uso. Encontraremos en capítulos posteriores que estas herramientas forman la base de una colección de herramientas que se usa para resolver una serie de problemas prácticos. Ésto será particularmente cierto cuando entremos en el shell scripting, donde estas herramientas mostrarán realmente su valor.

Para saber más

La web del proyecto GNU contiene muchas guías online de las herramientas que hemos visto en éste capítulo:

- Del paquete Coreutils:

http://www.gnu.org/software/coreutils/manual/coreutils.html#Output-of-entire-files
http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-sorted-files
http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-fields
http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-characters

- Del paquete Diffutils:
  http://www.gnu.org/software/diffutils/manual/html_mono/diff.html

- `sed`:
  http://www.gnu.org/software/sed/manual/sed.html
- `aspell`:
  http://aspell.net/man-html/index.html
- Hay otros muchos recursos online para `sed`, en particular:
  http://www.grymoire.com/Unix/Sed.html
  http://sed.sourceforge.net/sed1line.txt
- Prueba también a googlear “sed one liners”, “sed cheat sheets”

Crédito Extra

Hay unos pocos comandos interesantes más de manipulación de texto que merece la pena investigar. Entre ellos: `split` (divide archivos en trozos), `csplit` (divide archivos en trozos basados en contexto) y `sdiff` (unión de lado a lado de diferencias entre archivos.)

Formateando la salida

En este capítulo, continuamos viendo herramientas relacionadas con el texto, enfocándonos en programas que se usan para formatear salidas de texto, en lugar que cambiar el texto en sí. Estas herramientas se usan a menudo para preparar el texto para una eventual impresión, un caso que veremos en el próximo capítulo. Los programas que veremos en este capítulo incluyen:

- `nl` - Numera líneas
- `fold` - Limita cada línea a una longitud especificada
- `fmt` - Un formateador de texto simple
- `pr` - Prepara texto para imprimir
- `printf` - Formatea e imprime datos
- `groff` - Un sistema de formateo de documentos

Herramientas de formateo simple

Veremos algunas de las herramientas de formateo simple primero. Estas son mayormente programas de un único propósito, y poco sofisticadas en lo que hacen, pero pueden usarse para pequeñas tareas y como parte de entubados y scripts.

nl - Numera líneas

El programa nl es una herramienta más bien arcaica que se usa para realizar tareas simples. Numera las líneas. En su uso más simple, se parece a `cat -n`:

```
[me@linuxbox ~]$ nl distros.txt | head
1 SUSE 10.2 12/07/2006
2 Fedora 10 11/25/2008
3 SUSE 11.0 06/19/2008
4 Ubuntu 8.04 04/24/2008
5 Fedora 8 11/08/2007
6 SUSE 10.3 10/04/2007
7 Ubuntu 6.10 10/26/2006
8 Fedora 7 05/31/2007
9 Ubuntu 7.10 10/18/2007
10 Ubuntu 7.04 04/19/2007
```

Como `cat`, `nl` puede aceptar tanto múltiples archivos como argumentos de línea de comandos, o entrada estándar. Sin embargo, `nl` tiene un número de opciones y soporta una forma primitiva de etiquetas para permitir tipos de numeración más complejas.

`nl` soporta un concepto llamado "páginas lógicas" cuando numera. Ésto permite a `nl` resetear (empezar de nuevo) la secuencia numérica cuando está numerando. Usando opciones, es posible establecer el número inicial a un valor específico y, en limitada medida, su formato. Una página lógica está dividida en encabezado, cuerpo y pie. Dentro de cada una de estas secciones, el numerado de líneas puede ser reseteado y/o asignarle un estilo diferentes. Si le damos a `nl` múltiples archivos, los trata como una cadena simple de texto. Las secciones en la cadena de texto se indican con la presencia de algún tipo de etiqueta de estilo antiguo añadida al texto:

Tabla 21-1: Etiquetas nl

Etiqueta	Significado
`\:\:\:`	Comienzo del encabezado de la página lógica

`\:\:`	Comienzo del cuerpo de la página lógica
`\:`	Comienzo del pie de la página lógica

Cada etiqueta anterior debe aparecer sola en su propia línea. Tras procesar una etiqueta, `nl` la borra de la cadena de texto.

Aquí tenemos las opciones comunes de `nl`:

Tabla 21-2: Opciones comunes de nl

Opción	**Significado**
`-b style`	Establece la numeración del cuerpo a *style*, donde *style* es uno de los siguientes: `a` = numera todas las líneas `t` = numera sólo las líneas que no están en blanco. Es la opción por defecto. `n` = no `pregexp` = numera sólo las líneas que coinciden con la expresión regular *regexp*
`-f style`	Establece la numeración del pie a *style*. La opción por defecto es n (no)
`-h style`	Establece la numeración del encabezado a *style*. La opción por defecto es n (no)
`-i number`	Establece el incremento de numeración por página a *number*. La opción por defecto es uno.
`-n format`	Establece el formato de numeración a *format*, donde format es: `ln` = justificado a la izquierda, sin ceros delante. `rn` = justificado a la derecha, sin ceros delante. Es la opción por defecto. `rz` = justificado a la derecha, con ceros delante.
`-p`	No resetea la numeración de páginas al principio de cada página lógica.
`-s string`	Añade *string* al final de cada número de línea para crear un separador. La opción por defecto es un carácter de tabulación simple.
`-v number`	Establece el número de la primera línea de cada página lógica a *number*. La opción por defecto es uno.
`-w width`	Establece la anchura del campo del número de línea a *width*. La opción por defecto es 6.

Admitámoslo, probablemente no querremos numerar líneas tan a menudo, pero podemos usar `nl` para ver como podemos combinar múltiples herramientas para realizar tareas más complejas. Trabajaremos sobre nuestro ejercicio del capítulo anterior para producir un informe de distribuciones Linux. Como usaremos `nl`, será útil incluir sus etiquetas encabezado/cuerpo/pie. Para hacerlo, las añadiremos al script `sed` del último capítulo. Usando nuestro editor de texto, cambiaremos el script de la siguiente forma y lo guardaremos como `distros-nl.sed`:

```
# sed script to produce Linux distributions report
1 i\
\\:\\:\\:\
```

```
\
Linux Distributions Report\
\
Name Ver. Released\
---- ---- --------\
\\:\\:
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
\\:\
\
End Of Report
```

El script ahora inserta las etiquetas `nl` de páginas lógicas y añade un pie al final del informe. Fíjate que hora tenemos que duplicar las barras invertidas en nuestra etiqueta, porque normalmente se interpretan como un carácter de escape en `sed`.

A continuación, produciremos nuestro informe mejorado combinando `sort`, `sed` y `nl`:

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-nl.s
ed | nl
Linux Distributions Report
  Name   Ver. Released
  ----   ---- --------
 1 Fedora 5    2006-03-20
 2 Fedora 6    2006-10-24
 3 Fedora 7    2007-05-31
 4 Fedora 8    2007-11-08
 5 Fedora 9    2008-05-13
 6 Fedora 10   2008-11-25
 7 SUSE   10.1 2006-05-11
 8 SUSE   10.2 2006-12-07
 9 SUSE   10.3 2007-10-04
10 SUSE   11.0 2008-06-19
11 Ubuntu 6.06 2006-06-01
12 Ubuntu 6.10 2006-10-26
13 Ubuntu 7.04 2007-04-19
14 Ubuntu 7.10 2007-10-18
15 Ubuntu 8.04 2008-04-24
16 Ubuntu 8.10 2008-10-30
End Of Report
```

Nuestro informe es el resultado de nuestro entubado de comandos. Primero, ordenamos la lista por nombre de la distribución y versión (campos 1 y 2), luego procesamos el resultado con `sed`, añadiendo el encabezado del informe (incluyendo la etiqueta de página lógica de `nl`) y el pie. Finalmente, procesamos el resultado con `nl`, que, por defecto, sólo numera las líneas de la cadena de texto que pertenezcan a la sección cuerpo de la página lógica.

Podemos repetir el comando y experimentar con diferentes opciones de `nl`. Algunas interesantes son:

nl -n rz
y

nl -w 3 -s ' '

fold - Limita cada línea a una longitud especificada

Folding (plegado) es el proceso de cortar líneas de texto a una anchura especificada. Como nuestros otros comandos, `fold` acepta tanto uno o más archivos de texto como entrada estándar. Si mandamos a `fold` una cadena de texto simple, podemos ver como funciona:

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy
dog." | fold -w 12
The quick br
own fox jump
ed over the
lazy dog.
```

Aquí vemos a `fold` en acción. El texto enviado por el comando `echo` se divide en segmentos especificados por la opción -w. En este ejemplo, especificamos una anchura de línea de 12 caracteres. Si no se especifica anchura, la opción por defecto es 80 caracteres. Fíjate como las líneas se cortan independientemente de los límites de las palabras. Añadir la opción `-s` hace que `fold` corte la línea en el último espacio existente antes de que se alcance la longitud de la línea:

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy
dog." | fold -w 12 -s
The quick
brown fox
jumped over
the lazy

dog.
```

fmt - Un formateador de texto simple

El programa `fmt` también pliega el texto, pero un poco más. Acepta tanto archivos como entrada estándar y realiza formateo de párrafos en la cadena de texto. Básicamente, rellena y junta líneas de texto mientras que mantiene líneas en blanco y sangrías.

Para demostrarlo, necesitaremos algún texto. Tomemos algo de la página info de `fmt`:

```
`fmt' reads from the specified FILE arguments (or standard input
if none are given), and writes to standard output.



By default, blank lines, spaces between words, and indentation are
preserved in the output; successive input lines with different
indentation are not joined; tabs are expanded on input and
introduced
on output.
`fmt' prefers breaking lines at the end of a sentence, and tries
to avoid line breaks after the first word of a sentence or before
```

```
the
last word of a sentence. A "sentence break" is defined as either
the
end of a paragraph or a word ending in any of `.?!', followed by
two
spaces or end of line, ignoring any intervening parentheses or
quotes. Like TeX, `fmt' reads entire "paragraphs" before choosing
line breaks; the algorithm is a variant of that given by Donald E.
Knuth and Michael F. Plass in "Breaking Paragraphs Into Lines",
`Software--Practice & Experience' 11, 11 (November 1981), 1119-
1184.
```

Copiaremos este texto en nuestro editor de texto y guardaremos el archivo como `fmt-info.txt`. Ahora, digamos que queremos reformatear este texto para ajustarlo a columnas de cincuenta caracteres de ancho. Podríamos hacerlo procesando el archivo con `fmt` y la opción `-w`:

```
[me@linuxbox ~]$ fmt -w 50 fmt-info.txt | head
`fmt' reads from the specified FILE arguments
(or standard input if
none are given), and writes to standard output.



By default, blank lines, spaces between words,
and indentation are
preserved in the output; successive input lines
with different indentation are not joined; tabs
are expanded on input and introduced on output.
```

Bien, es un resultado poco manejable. Quizá deberíamos leer el texto en realidad, ya que explica lo que está pasando:

Por defecto, las líneas en blanco, los espacios entre palabras y las sangrías se mantienen en la salida; las líneas de entrada sucesivas con diferentes sangría no se añaden; las tabulaciones se expanden en la entrada y se pasan a la salida.

Así que, `fmt` mantiene la sangría de la primera línea. Afortunadamente, `fmt` proporciona una opción para corregir ésto:

```
[me@linuxbox ~]$ fmt -cw 50 fmt-info.txt
`fmt' reads from the specified FILE arguments
(or standard input if none are given), and writes
to standard output.
By default, blank lines, spaces between words,
and indentation are preserved in the output;
successive input lines with different indentation
are not joined; tabs are expanded on input and
introduced on output.
`fmt' prefers breaking lines at the end of a
sentence, and tries to avoid line breaks after
the first word of a sentence or before the
last word of a sentence. A "sentence break"
is defined as either the end of a paragraph
```

```
or a word ending in any of `.?!', followed
by two spaces or end of line, ignoring any
intervening parentheses or quotes. Like TeX,
`fmt' reads entire "paragraphs" before choosing
line breaks; the algorithm is a variant of
that given by Donald E. Knuth and Michael F.
Plass in "Breaking Paragraphs Into Lines",
`Software--Practice & Experience' 11, 11
(November 1981), 1119-1184.
```

Mucho mejor. Añadiendo la opción `-c`, ahora tenemos el resultado deseado.

`fmt` tiene algunas opciones interesantes:

Tabla 21-3: Opciones fmt

Opción	Descripción
`-c`	Opera en *modo margen de corona*. Esto preserva la tabulación de las primeras dos líneas de un párrafo. Las siguientes líneas se alinean con la tabulación de la segunda línea.
`-p` *`string`*	Solo formatea aquellas líneas que comienzan con el prefijo *string*. Tras formatear, el contenido de *string* se coloca como prefijo de cada línea reformateada. Esta opción puede usarse para formatear texto en comentarios de código fuente. Por ejemplo, cada lenguaje de programación o archivo de configuración que use un carácter `"#"` para delimitar un comentario podría formatearse especificando `-p '# '` de forma que sólo los comentarios se formatearán. Mira el ejemplo a continuación.
`-s`	Modo sólo-división. En este modo, las líneas sólo se dividirán para ajustarse al ancho especificado de columna. Las líneas cortas no se unirán para rellenar líneas. Este modo es útil cuando formateamos texto como código donde no queremos que se unan las líneas.
`-u`	Realiza espaciado uniforme. Ésto aplicará el estilo tradicional de "máquina de escribir" al texto. Esto significa que se aplica un espacio entre palabras y dos espacios entre frases. Este modo es útil para eliminar la "justificación", o sea, el texto que ha sido inflado con espacios para forzar el alineamiento tanto en el margen derecho como izquierdo.
`-w` *`ancho`*	Formatea el texto para ajustarlo a un *ancho* de columna en caracteres. La opción por defecto es 75 caracteres. Nota: `fmt` en realidad formatea líneas ligeramente más cortas que la anchura especificada para permitir balanceo de líneas.

La opción `-p` es particularmente interesante. Con ella, podemos formatear fragmentos seleccionados de un archivo, siempre que las líneas a formatear comiencen con la misma secuencia de caracteres. Muchos lenguajes de programación usan el símbolo de la libra (#) para indicar el comienzo de un comentario y así pueden ser formateados usando esta opción. Creemos un archivo que simule un programa que use comentarios:

```
[me@linuxbox ~]$ cat > fmt-code.txt
# This file contains code with comments.
```

```
# This line is a comment.
# Followed by another comment line.
# And another.



This, on the other hand, is a line of code.
And another line of code.
And another.
```

Nuestro archivo de ejemplo contiene comentarios que comienzan con la cadena `"# "` (un # seguido por un espacio) y líneas de código que no. Ahora, usando `fmt`, podemos formatear los comentarios y dejar el código intacto:

```
[me@linuxbox ~]$ fmt -w 50 -p '# ' fmt-code.txt
# This file contains code with comments.



# This line is a comment. Followed by another
# comment line. And another.



This, on the other hand, is a line of code.
And another line of code.
And another.
```

Fíjate que las líneas de comentarios contiguas se unen, mientras que las líneas en blanco y las que no comienzan con el prefijo especificado se conservan.

pr - Formatea texto para imprimir

El programa `pr` se usa para *paginar* texto. Cuando imprimimos texto, a menudo es deseable separar las páginas de la salida con varias líneas en blanco, para darle un margen superior e inferior a cada página. Mas adelante, este espacio en blanco puede usarse para insertar un encabezado y un pie en cada página.

Probaremos `pr` formateando el archivo `distros.txt` en una serie de páginas muy cortas (sólo las dos primeras páginas se muestran):

```
[me@linuxbox ~]$ pr -l 15 -w 65 distros.txt



2008-12-11 18:27    distros.txt     Page 1
SUSE    10.2      12/07/2006
Fedora  10        11/25/2008
SUSE    11.0      06/19/2008
```

```
Ubuntu  8.04      04/24/2008
Fedora  8         11/08/2007




2008-12-11 18:27    distros.txt     Page 2




SUSE    10.3      10/04/2007
Ubuntu  6.10      10/26/2006
Fedora  7         05/31/2007
Ubuntu  7.10      10/18/2007
Ubuntu  7.04      04/19/2007
```

En este ejemplo, empleamos la opción `-l` (para la longitud de página) y la opción `-w` (ancho de página) para definir una "página" de 65 columnas de ancho y 15 líneas de largo. `pr` pagina el contenido del archivo `distros.txt`, separando cada página con varias líneas en blanco y crea un encabezado por defecto que contiene la hora de modificación del archivo y el número de página. El programa `pr` ofrece muchas opciones para controlar la plantilla de página. Echaremos un vistazo más a fondo en el siguiente capítulo

printf - Formatea e imprime datos

Al contrario que otros comandos de este capítulo, el comando `printf` no se usa para entubados (no acepta entrada estándar) ni se usa frecuentemente en la línea de comandos (se usa mayormente en scripts). Entonces ¿Por qué es tan importante? Porque se usa mucho.

`printf` (de la frase "print formatted" - impresión formateada) fue desarrollado originalmente para el lenguaje de programación C y ha sido implementado en muchos lenguajes de programación incluyendo el shell. De hecho, en `bash`, `printf` está incluido.

`printf` funciona así:

printf "*formato*" *argumentos*

Se le da al comando una cadena que contiene una descripción del formato que se aplica a una lista de argumentos. El resultado formateado se envía a la salida estándar. Aquí hay un ejemplo trivial:

```
[me@linuxbox ~]$ printf "I formatted the string: %s\n" foo
I formatted the string: foo
```

La cadena de formato puede contener texto literal (como "He formateado la cadena:"), secuencias de espcape (como `\n`, un caracter de nueva línea), y secuencias que comiencen con el carácter %, que se llaman especificaciones de conversion. En el ejemplo anterior, la especificación de conversión `%s` se usa para formatear la cadena "foo" y colocarla en la salida del comando. Aquí está de nuevo:

```
[me@linuxbox ~]$ printf "I formatted '%s' as a string.\n" foo
I formatted 'foo' as a string.
```

Como podemos ver, la especificación de conversión `%s` se reemplaza por la cadena "foo" en la salida del comando. La conversión s se usa para formatear cadenas de datos. Hay otros especificadores para otros tipos de datos. Esta tabla lista los tipos de datos usados más frecuentemente:

Tabla 21-4: Especificadores de tipos de datos frecuentes en printf

Especificador	Descripción
`-d`	Formatea un número como un entero decimal con signo.
`f`	Formatea y manda a la salida un número de coma flotante.
`o`	Formatea un entero como un número octal.
`s`	Formatea una cadena.
`x`	Formatea un entero como número hexadecimal usando a-f en minúsculas donde sea necesario.
`X`	Lo mismo que `x` pero usa letras mayúsculas.
`%`	Imprime un símbolo `%` literal (p.ej.: especificando "%%")

Probaremos el efecto de cada especificador de conversión en la cadena "380":

```
[me@linuxbox ~]$ printf "%d, %f, %o, %s, %x, %X\n" 380 380 380 380 380 380
380, 380.000000, 574, 380, 17c, 17C
```

Como hemos usado seis especificadores de conversión, tenemos que añadir también seis argumentos a `printf` para el proceso. Los seis resultados muestran el efecto de cada especificador.
Varios componentes opciones pueden añadirse al especificador de conversión para ajustar la salida. Una especificación de conversión completa consiste en lo siguiente:

```
%[flags][width][.precision]conversion_specification
```

Múltiples componentes opcionales, cuando se usan, deben aparecer en el orden indicado arriba para que sean interpretados correctamente. Aquí tenemos una descripción de cada uno:

Tabla 21-5: Componentes de especificación de conversión de printf

Componente	Descripción
`flags`	Hay diferentes flags:
	# - Usa el "formato alternativo" para la salida. Esto varía según el tipo de dato. Para conversión `o` (número octal), la salida es precedida con `0`. Para conversiones `x` y `X` (números hexadecimales), la salida es precedida con `0x` o `0X` respectivamente.
	`0` - (cero) Rellena la salida con ceros. Ésto significa que el campo se rellenará con ceros delante, como en "000380".

	`-` - (guión) Alinea la salida a la izquierda. Por defecto, `printf` alinea a la derecha la salida.
	`' '` - (espacio) Produce un espacio delante para números positivos.
	`+` - (signo más) Pone signo a los números positivos. Por defecto, `printf` sólo pone signo a los números negativos.
width	Un número especificando la anchura mínima del campo.
. precission	Para número de coma flotante, especifica el número de dígitos de precisión a mostrar tras la coma. En conversión de cadenas, precisión especifica el número de caracteres a mostrar.

Aquí tenemos algunos ejemplos de diferentes formatos en acción:

Tabla 21-6: Ejemplos de especificación de conversión en print

Argumento	**Formato**	**Resultado**	**Notas**
380	"%d"	380	Formateo simple de un entero.
380	"%#x"	0x17c	Entero formateado como un número hexadecinal usando la etiqueta "formato alternativo".
380	"%05d"	00380	Entero formateado con ceros delante (rellenado) y una anchura mínima de cinco caracteres.
380	"%05.5f"	380,00000	Número formateado como número en coma flotante con rellenado y cinco posiciones decimales de precisión. Como la anchura de campo mínima especificada (5) es menor de la anchura real del número formateado, el rellenado no tiene efecto.
380	"%010.5f"	0380,00000	Incrementando la anchura mínima del campo a 10 el rellenado es ahora visible.
380	"%+d"	+380	La etiqueta + pone signo a un número positivo.
380	"%-d"	380	La etiqueta - alinea el formato a la izquierda.
abcdefghijk	"%5s"	abcedfghijk	Una cadena formateada con un ancho de campo mínimo.
abcdefghijk	"%.5s"	abcde	Aplicando precisión a una cadena, se trunca.

De nuevo, `printf` se usa mayormente en scripts donde se emplea para formatear datos tabulares, en lugar de en la línea de comandos directamente. Pero si que podemos ver como puede usarse para

resolver varios problemas de formateo. Primero, obtengamos varios campos separados por caracteres tabulador:

```
[me@linuxbox ~]$ printf "%s\t%s\t%s\n" str1 str2 str3
str1 str2 str3
```

Insertando `\t` (la secuencia de escape para un tabulador), logramos el efecto deseado. A continuación, algunos número con formateo especial:

```
[me@linuxbox ~]$ printf "Line: %05d %15.3f Result: %+15d\n" 1071
3.14156295 32589
Line: 01071 3.142 Result: +32589
```

Esto muestra el efecto del ancho mínimo de campo en el espacio de los campos. O cómo formatear una pequeña página web:

```
[me@linuxbox ~]$ printf "<html>\n\t<head>\n\t\t<title>%s</title>\n
\t</head>\n\t<body>\n\t\t<p>%s</p>\n\t</body>\n</html>\n" "Page
Tit
le" "Page Content"
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <p>Page Content</p>
  </body>

</html>
```

Sistemas de formateo de documentos

Hasta ahora, hemos examinado las herramientas de formato de texto simple. Son buenas para tareas simples y pequeñas, pero ¿que pasa con grandes trabajos? Una de las razones que hizo que Unix llegara a ser un sistema operativo popular entre usuarios técnicos y científicos (ademas de proporcionar una multitarea potente y entorno multiusuario para todo tipo de desarrollo de software) es que ofrece herramientas que pueden usarse para producir muchos tipos de documentos, particularmente publicaciones científicas y académicas. De hecho, como la documentación GNU describe, la preparación de documentos era instrumental para el desarrollo de Unix:

La primera versión de UNIX fue desarrollada en un PDP-7 que estaba cercano a los laboratorios Bell. En 1971 los desarrolladores querían conseguir un PDP-11 para futuros trabajos en el sistema operativo. Para justificar el coste de este sistema, propusieron e implementaron un sistema de formateo de documentos para la división de patentes de AT&T. Este primer programa de formateo fue una reimplementación del McIllroy's 'roff', escrito por J. F. Ossanna.

Dos principales familias de formateadores de documentos dominaban el terreno. aquellos que descendían del programa `roff` original, incluyendo `nroff` y `troff`, y aquellos basados en el sistema de escritura T$_{E}$X (pronunciado "tek") de Donald Knuth. Y sí, la "E" caída en el centro es parte del nombre.

El nombre "roff" deriva del término "run off" como en, "I'll run off a copy for you. - Imprimiré una

copia para tí". El programa `nroff` se usa para formatear documentos para salida a dispositivos que usan fuentes mono espacio, como los terminales de caracteres e impresoras estilo máquina de escribir. En el momento de su introducción, incluía casi todos los dispositivos de impresión conectados a ordenadores. El posterior programa `troff` formatea documentos para salida a tipógrafos, dispositivos usados para producir escritura "preparada para cámara" para impresión comercial. La mayoría de las impresoras de ordenadores hoy en día pueden simular la salida de los tipógrafos. La familia `roff` también incluye algunos otros programas que se usan para preparar porciones de documentos. Estos incluyen `eqn` (para ecuaciones matemáticas) y `tbl` (para tablas).

El sistema T_EX (en su forma estable) apareció por primera vez en 1989 y ha, en algún grado, desplazado a `troff` como la herramienta elegida para salida a tipógrafos. No veremos T_EX aquí, debido a su complejidad (hay libros enteros sobre él) y al hecho de que no viene instalada por defecto en sistemas Linux modernos.

Consejo: Para aquellos interesados en instalar T_EX, prueba el paquete `texlive` que puede encontrarse en la mayoría de los repositorios de las distribuciones, y el editor de contenido gráfico LyX.

groff

`groff` es una colección de programas que contienen la implementación GNU de `troff`. También incluye un script que se usa para emular `nroff` y el resto de la familia `roff` también.

Mientras que `roff` y sus descendientes se usan para crear documentos formateados, lo hacen de una forma algo extraña para los usuarios modernos. La mayoría de los documentos de hoy en día se producen usando procesadores de texto que pueden realizar tanto la composición como el formateado de un documento en un único paso. Antes de la llegada de los procesadores de texto gráficos, los documentos a menudo se producían en un proceso en dos pasos que incluía el uso de un editor de texto para realizar la composición, y un procesador, como `troff`, para realizar el formateo. Las instrucciones para el programa de formateo estaban incluidas en el texto compuesto a través del uso de un lenguaje de etiquetas. La analogía moderna para este tipo de procesos es la página web, que se crea usando un editor de texto de algún tipo y luego se renderiza usando un navegador web con HTML como lenguaje de etiquetas para describir el aspecto final de la página.

No vamos a ver `groff` en su totalidad, ya que muchos elementos de su lenguaje de etiquetas tienen que ver con detalles de tipografía muy arcaicos. En su lugar nos concentraremos en uno de sus macro paquetes que siguen usando ampliamente. Los macro paquetes concentran muchos de sus comandos de bajo nivel en pequeñas colecciones de comandos de alto nivel que hace el uso de `groff` mucho más sencillo.

Por un momento, consideremos la humilde man page. Se encuentra en el directorio `/usr/share/man` como un archivo de texto comprimido con `gzip`. Si fuéramos a examinar su contenido descomprimido, veríamos lo siguiente (se muestra la man page de ls en su sección 1):

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | head
.\" DO NOT MODIFY THIS FILE! It was generated by help2man 1.35.
.TH LS "1" "April 2008" "GNU coreutils 6.10" "User Commands"
.SH NAME
ls \- list directory contents
.SH SYNOPSIS
```

```
.B ls
[\fIOPTION\fR]... [\fIFILE\fR]...
.SH DESCRIPTION
.\" Add any additional description here
.PP
```

Comparada a la man page en su presentación normal, podemos empezar a ver una correlación entre el lenguaje de marcas y su resultado:

```
[me@linuxbox ~]$ man ls | head
LS(1) User Commands LS(1)
NAME
ls - list directory contents
SYNOPSIS
ls [OPTION]... [FILE]...
```

La razón por la que esto es interesante es que las man pages están renderizadas por `groff`, usando el macro paquete `man-doc`. De hecho, podemos simular el comando `man` con el siguiente entubado:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc
-T ascii | head
LS(1) User Commands LS(1)
NAME
ls - list directory contents
SYNOPSIS
ls [OPTION]... [FILE]...
```

Aquí usamos el programa `groff` con las opciones configuradas para especificar el macro paquete `mandoc` y el controlador de salida para ASCII. `groff` puede producir salida en varios formatos. Si no se especifica ningún formato, la salida por defecto es PostScript:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc
|
head
%!PS-Adobe-3.0
%%Creator: groff version 1.18.1
%%CreationDate: Thu Feb 5 13:44:37 2009
%%DocumentNeededResources: font Times-Roman
%%+ font Times-Bold
%%+ font Times-Italic
%%DocumentSuppliedResources: procset grops 1.18 1
%%Pages: 4
%%PageOrder: Ascend
%%Orientation: Portrait
```

Hemos mencionado brevemente PostScript en el capítulo anterior, y lo haremos de nuevo en el próximo capítulo. PostScript es un lenguaje de descripción de página que se usa para describir el contenido de una página impresa en un dispositivo tipo teletipo. Si tomamos la salida de nuestro comando y lo almacenamos en un archivo (asumiendo que estamos usando un entorno gráfico con un directorio `Desktop`):

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc
>
~/Desktop/foo.ps
```

Un icono del archivo de salida aparecerá en el escritorio. Haciendo doble clic en el icono, se abrirá un visor de páginas y mostrará el archivo en su forma renderizada:

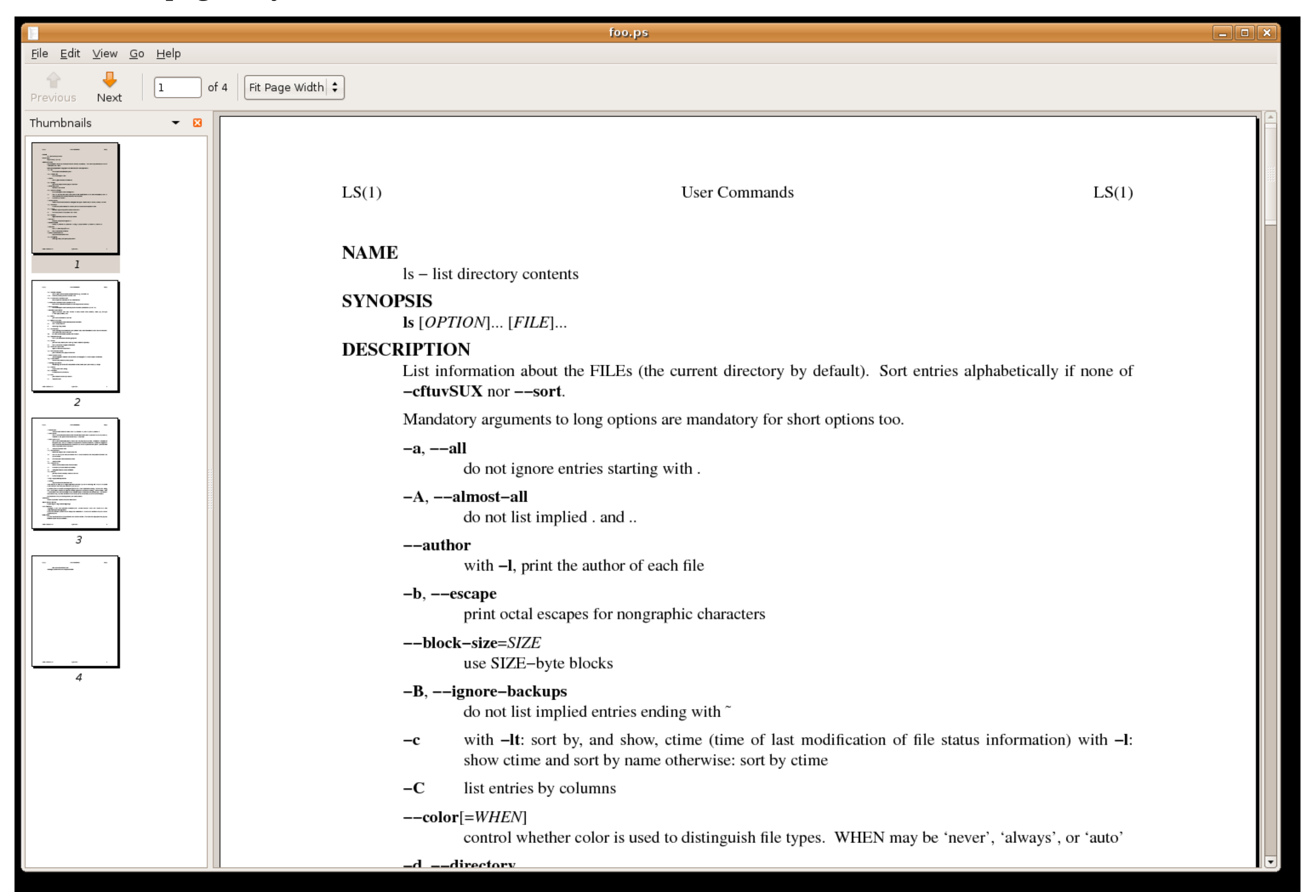


Figura 4: Viendo la salida PostScript con un visor de páginas en GNOME

Lo que vemos es ¡una bonita man page de `ls`! De hecho, es posible convertir el archivo PostScript a PDF (*Portable Document Format - Formato de Documento Portable*) con este comando:

```
[me@linuxbox ~]$ ps2pdf ~/Desktop/foo.ps ~/Desktop/ls.pdf
```

El programa `ps2pdf` es parte del paquete `ghostscript`, que está instalado en la mayoría de los sistemas Linux que soportan impresión:

Consejo: Los sistemas Linux a menudo incluyen muchos programas de línea de comandos para conversión de formatos de archivos. A menudo les nombra usando la convención *format2format*. Prueba a usar el comando `ls /usr/bin/*[[:alpha:]]2[[:alpha:]]*` para identificarlos. Prueba también a buscar programas llamados *format***to***format*.

Para nuestro último ejercicio con `groff`, revisitaremos nuestro viejo amigo `distros.txt` una vez más. Esta vez, usaremos el programa `tbl` que se usa para formatear tablas para imprimir nuestra lista de distribuciones Linux. Para hacerlo, vamos a usar nuestro anterior script de `sed` para añadir etiquetas a una cadena de texto que luego enviaremos a `groff`.

Primero, necesitamos modificar nuestro script `sed` para añadir las peticiones necesarios que requiere `tbl`. Usando un editor de texto, cambiaremos `distros.sed` a lo siguiente:

```
# sed script to produce Linux distributions report
1 i\
.TS\
center box;\
cb s s\
cb cb cb\
l n c.\
Linux Distributions Report\
=\
Name Version Released\
_
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
.TE
```

Fíjate que para que el script funcione correctamente, hay que tener cuidado y comprobar que las palabras "Name Version Released" está separadas por tabuladores, no por espacios. Guardaremos el archivo resultante como `distros-tbl.sed.` `tbl` usa las etiquetas `.TS` y `.TE` para comenzar y finalizar la tabla. Las filas que siguen a la etiqueta `.TS` definen las propiedades globales de la tabla que, en nuestro ejemplo, está centrada horizontalmente en la página y rodeada por una caja. Las restantes líneas de la definición describen el estilo de cada fila de la tabla. Ahora, si ejecutamos nuestro entubado generador de reportes de nuevo con el nuevo script de `sed`, tendremos lo siguiente:

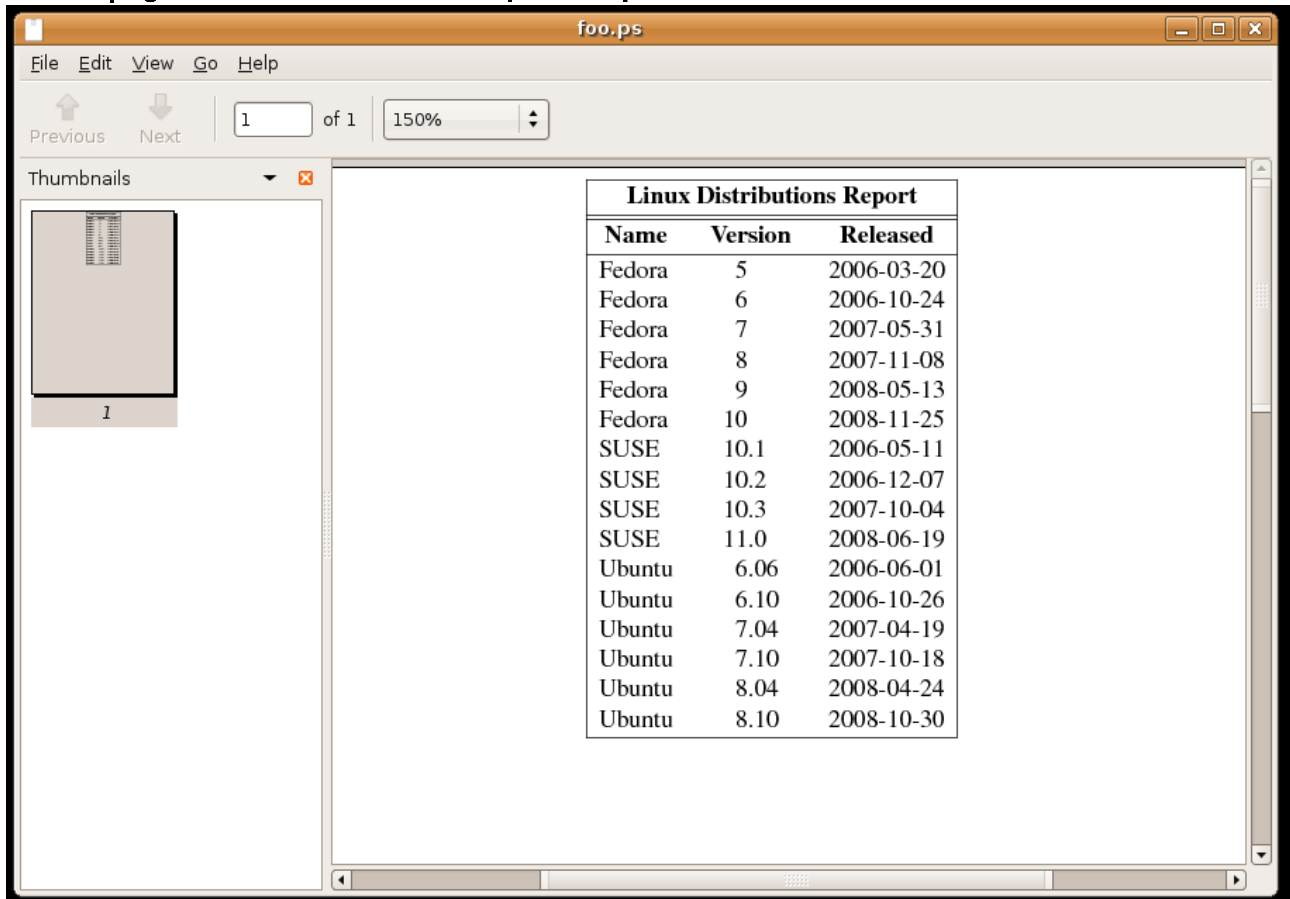
```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl
.sed | groff -t -T ascii 2>/dev/null
+------------------------------+
| Linux Distributions Report |
+------------------------------+
| Name Version Released |
+------------------------------+
|Fedora     5       2006-03-20 |
|Fedora     6       2006-10-24 |
|Fedora     7       2007-05-31 |
|Fedora     8       2007-11-08 |
|Fedora     9       2008-05-13 |
|Fedora    10       2008-11-25 |
|SUSE      10.1     2006-05-11 |
|SUSE      10.2     2006-12-07 |
|SUSE      10.3     2007-10-04 |
|SUSE      11.0     2008-06-19 |
|Ubuntu     6.06    2006-06-01 |
|Ubuntu     6.10    2006-10-26 |
|Ubuntu     7.04    2007-04-19 |
|Ubuntu     7.10    2007-10-18 |
|Ubuntu     8.04    2008-04-24 |
|Ubuntu     8.10    2008-10-30 |
+------------------------------+
```

Añadiendo la opción `-t` a `groff` le ordenamos que pre-procese la cadena de texto con `tbl`. De igual forma, la opción `-T` se usa para salida ASCII en lugar del medio de salida por defecto,

PostScript.

El formato de la salida es el mejor que podemos esperar si estamos limitados por las capacidades de una pantalla de terminal o una impresora tipo máquina de escribir. Si especificamos salida PostScript y vemos gráficamente la salida resultante, tenemos un resultado más satisfactorio:

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl
.sed | groff -t > ~/Desktop/foo.ps
```



Linux Distributions Report		
Name	**Version**	**Released**
Fedora	5	2006-03-20
Fedora	6	2006-10-24
Fedora	7	2007-05-31
Fedora	8	2007-11-08
Fedora	9	2008-05-13
Fedora	10	2008-11-25
SUSE	10.1	2006-05-11
SUSE	10.2	2006-12-07
SUSE	10.3	2007-10-04
SUSE	11.0	2008-06-19
Ubuntu	6.06	2006-06-01
Ubuntu	6.10	2006-10-26
Ubuntu	7.04	2007-04-19
Ubuntu	7.10	2007-10-18
Ubuntu	8.04	2008-04-24
Ubuntu	8.10	2008-10-30

Figura 5: Viendo la tabla finalizada

Resumiendo

Dado que el texto es tan fundamental para el carácter de los sistemas operativos como-Unix, tiene sentido que haya muchas herramientas que se usan para manipular y formatear texto. Como hemos visto, ¡Las hay! Las herramientas de formateo simple como `fmt` y `pr` encontrarán muchos usos en scripts que producen documentos cortos, mientras que `groff` (y sus amigos) pueden usarse para escribir libros. Nunca escribiremos un paper técnico usando herramientas de línea de comandos (¡Aunque hay mucha gente que lo hace!), pero es bueno saber que podríamos.

Para saber más

- La guía de usuario de `groff`

http://www.gnu.org/software/groff/manual/

- *Redactando Papers con nroff usando* `-me`:

http://docs.freebsd.org/44doc/usd/19.memacros/paper.pdf

- *Manual de referencia de* `-me`*:*

http://docs.freebsd.org/44doc/usd/20.meref/paper.pdf

- *Tbl – Un programa para formatear tablas:*

http://plan9.bell-labs.com/10thEdMan/tbl.pdf

- *Y, por supuesto, mira los siguientes artículos de la Wikipedia:*

http://en.wikipedia.org/wiki/TeX
http://en.wikipedia.org/wiki/Donald_Knuth
http://en.wikipedia.org/wiki/Typesetting

Impresión

Tras pasar el último par de capítulos manipulando texto, es hora de poner ese texto en un papel. En este capítulo, veremos las herramientas de línea de comandos que se usan para imprimir archivos y controlar la operación de impresión. No veremos como configurar la impresión, ya que varía según la distribución y normalmente se configura automáticamente durante la instalación. Fíjate que necesitaremos un impresora funcionando y configurada para realizar los ejercicios de este capítulo. Veremos los siguientes comandos:

- `pr` - Convierte archivos de texto para imprimir
- `lpr` - Imprime archivos
- `a2ps` - Formatea archivos para imprimir en una impresora PostScript
- `lpstat` - Muestra la información de estado de la impresora
- `lpq` - Muestra el estado de la cola de impresión
- `lprm` - Cancela trabajos de impresión

Una breve historia de la impresión

Para entender completamente las características de impresión que encontramos en los sistemas operativos como-Unix, tenemos que aprender antes algo de historia. Imprimir en los sistemas como-Unix nos lleva de vuelta al origen del sistema operativo en sí. En aquellos días, las impresoras y su uso eran muy diferentes a hoy.

Imprimiendo en los tiempos oscuros

Como los propios ordenadores, las impresoras en la era pre-PC tendían a ser grandes, caras y centralizadas. El típico usuario de ordenador de 1980 trabajaba en un terminal conectado a un ordenador que estaba a cierta distancia. La impresora se encontraba cerca del ordenador y estaba bajo los ojos vigilantes de los operadores del ordenador.

Cuando las impresoras eran caras y centralizadas, como lo eran a menudo en los primeros tiempos de Unix, era una práctica común para muchos usuarios compartir una impresora. Para identificar los trabajos de impresión pertenecientes a un usuario en particular, se imprimía a menudo una *página cabecera* mostrando el nombre del usuario al principio de cada trabajo de impresión. El equipo de soporte del ordenador cargaría entonces un carro con los trabajos de impresión del día y los enviaría a cada usuario individual.

Impresoras basadas en caracteres

La tecnología de impresión de los 80 era muy diferente en dos aspectos. Primero, las impresoras de dicho periodo eran casi todas *impresoras de impacto*. Las impresoras de impacto usan un

mecanismo mecánico que golpea una cinta contra el papel para formar impresiones de caracteres en la página. Dos de las tecnologías populares de aquella época eran la impresión de *margarita* y la impresión por *matriz de puntos*.

La segunda, y más importante característica de las primeras impresoras era que las impresoras usaban una colección limitada de caracteres que era intrínseca al propio dispositivo. Por ejemplo, una impresora de margarita sólo podía imprimir los caracteres que estaban moldeados en los pétalos de la margarita. Esto hacía que las impresoras fueran casi como máquinas de escribir de alta velocidad. Como la mayoría de máquinas de escribir, imprimían usando fuentes mono espacio (de anchura fija). Esto significa que cada carácter tiene la misma anchura. La impresión se realizaba en posiciones fijas de la página, y el área imprimible de una página contenía un número fijo de caracteres. La mayoría de las impresoras imprimían diez caracteres por pulgada (CPI) horizontalmente y seis líneas por pulgada (LPI) verticalmente. Usando este esquema, una hoja de papel tamaño US-letter tenía 85 caracteres de ancho y 66 líneas de ancho. Teniendo en cuenta un pequeño margen a cada lado, se consideraba que 80 caracteres era el ancho máximo de una línea impresa. Esto explica porque las pantallas de terminales (y nuestros emuladores de terminal) tienen normalmente 80 caracteres de ancho. Proporciona una vista WYSIWYG (*What you see is what you get* - Lo que ves es lo que obtendrás) de la salida impresa, usando una fuente mono espacio.

Los datos son enviados a una impresora tipo máquina de escribir en una cadena simple de bytes que contiene los caracteres a imprimir. Por ejemplo, para imprimir una "a", se envía el código ASCII de carácter 97. Además, los códigos de control ASCII con número bajos proporcionan un medio de mover el carro de la impresora y el papel, usando códigos para retorno de carro, salto de línea, salto de página, etc. Usando los códigos de control, es posible conseguir efectos de texto limitados, como negrita, haciendo a la impresora imprimir un carácter, un paso atrás, e imprimir el carácter de nuevo para conseguir una impresión más oscura en la página. En realidad podemos comprobar ésto si usamos `nroff` para renderizar una man page y ver la salida usando `cat -A`:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | nroff -man |
cat
-A | head
LS(1) User Commands LS(1)
$
$
$
N^HNA^HAM^HME^HE$
ls - list directory contents$
$
S^HSY^HYN^HNO^HOP^HPS^HSI^HIS^HS$
l^Hls^Hs [_^HO_^HP_^HT_^HI_^HO_^HN]... [_^HF_^HI_^HL_^HE]...$
```

Los caracteres `^H` (Control-h) son dos pasos atrás usados para crear el efecto negrita. De igual forma, podemos ver una secuencia paso atrás/subrayado para producir un subrayado.

Impresoras gráficas

El desarrollo de GUIs condujo a grandes cambios en la tecnología de impresión. Como los ordenadores evolucionaron a pantallas más basadas en imágenes, la impresión evolucionó de técnicas basadas en caracteres a otras basadas en gráficos. Esto fue ayudado por la llegada de impresoras láser de bajo costo que, en lugar de imprimir caracteres fijos, podían imprimir diminutos puntos en cualquier parte del área imprimible de la página. Esto hizo posible la impresión de

fuentes proporcionales (como las usadas en los teletipos) e incluso fotografías y diagramas de alta resolución.

Sin embargo, evolucionar de un esquema basado en caracteres a otro basado en gráficos requería un desafío tecnológico formidable. Aquí está el por qué: El número de bytes necesarios para llenar una página usando una impresora basada en caracteres podía calcularse de esta forma (asumiendo 60 líneas por página cada una con 80 caracteres):

60 X 80 = 4800 bytes

En comparación, una impresora de 300 puntos por pulgada (DPI) (asumiendo un área de impresión por página de 8 por 10 pulgadas) requiere:

(8 X 300) X (10 X 300) / 8 = 900000 bytes

Muchas de las lentas redes de PC simplemente no podían soportar el casi un megabyte de datos requeridos para imprimir una página completa en una impresora láser, así que estaba claro que se necesitaba una invención ingeniosa.

Esa invención resultó ser el *lenguaje de descripción de página* (PDL - page descriptión language). Un lenguaje de descripción de página es un lenguaje de programación que describe el contenido de una página. Básicamente dice, "ve a esta posición, dibuja del carácter 'a' en Helvetica de 10 puntos, ve a esta posición..." hasta que todo lo que haya en la página haya sido descrito. El primer gran PDL fue *PostScript* de Adobe Systems, que todavía está en uso hoy en día. El lenguaje PostScript es un lenguaje de programación completo adaptado a tipografía y otros tipos de gráficos e imágenes. Incluye soporte integrado para 35 fuentes estándar de gran calidad, más la capacidad de aceptar definiciones de fuentes adicionales al ejecutarse. De principio, el soporte para PostScript está incluido en las propias impresoras. Ésto solucionaba el problema de la transmisión de datos. Aunque el típico programa PostScript era muy verbal en comparación a la simple cadena de caracteres de las impresoras basadas en caracteres, era mucho más pequeño que el número de bytes requeridos para representar la página impresa completa.

Una *impresora PostScript* acepta un programa PostScript como entrada. La impresora cuenta con su propio procesador y memoria (haciendo que a veces la impresora sea más potente que el ordenador al que está conectada) y ejecuta un programa especial llamado *intérprete PostScript*, que lee el programa PostScript entrante y *renderiza* el resultado en la memoria interna de la impresora, que forma el patrón de bits (puntos) que serán transferidos al papel. El nombre genérico de este proceso de renderizar algo a un gran patrón de puntos (llamado *bitmap*) es *raster image processor* o RIP.

Según pasaron los años, tanto los ordenadores como las redes se hicieron mucho más rápidas. Esto permitió mover el RIP del la impresora al ordenador, que, a cambio, permitió que las impresoras de gran calidad fueran mucho menos caras.

Muchas impresoras hoy todavía aceptan cadenas basadas en caracteres, pero muchas impresoras de bajo coste no. Confían en el RIP del ordenador para que les proporcione una cadena de bits para imprimir como puntos. Todavía hay algunas impresoras PostScript, también.

Imprimiendo con Linux

Los sistemas Linux modernos emplean dos paquetes de software para realizar y gestionar impresiones. El primero, CUPS (Common Unix Printing System - Sistema Común de Impresión Unix) proporciona controladores de impresión y gestión de trabajos de impresión, y el segundo,

Ghostscript, un interprete PostScript, actúa como un RIP.

CUPS gestiona impresoras creando y manteniendo colas de impresión. Como vimos en nuestra lección de historia anterior, la impresión Unix fue originalmente diseñada para manejar una impresora central compartida por múltiples usuarios. Como las impresoras son lentas por naturaleza, comparadas con los ordenadores que las alimentan, los sistemas de impresión necesitan un modo de programar múltiples trabajos de impresión y mantener las cosas organizadas. CUPS también tiene la capacidad de reconocer diferentes tipos de datos (dentro de lo razonable) y puede convertir archivos a una forma imprimible.

Preparando archivos para imprimir

Como usuarios de la línea de comandos, estamos principalmente interesados en imprimir texto, aunque es realmente posible imprimir otros formatos de datos también.

pr - Convierte archivos de texto para imprimir

Vimos un poco sobre `pr` en el capítulo anterior. Ahora examinaremos algunas de sus muchas opciones usadas junto a la impresión. En nuestra historia de la impresión, vimos como las impresoras basadas en caracteres usan fuentes mono espacio, que producen números fijos de caracteres por línea y líneas por página. `pr` se usa para ajustar texto para encajarlo en un tamaño específico de página, con encabezados y márgenes opcionales. Aquí hay un sumario de sus opciones más comúnmente usadas:

Tabla 22-1: Opciones comunes de pr

Opción	Descripción
`+`*`first`*`[:`*`last`*`]`	Genera un rango de páginas que comienza con *first* y , opcionalmente termina con *last*.
`-`*`columns`*	Organiza el contenido de la página en el número de columnas especificado por *columns*.
`-a`	Por defecto, la salida multicolumna se lista verticalmente. Añadiendo la opción -a (across), el contenido se lista horizontalmente.
`-d`	Salida a doble espacio.
`-D "`*`format`*`"`	Formatea la fecha mostrada en los encabezados de página usando *format*. Mira la man page del comando `date` para ver una descripción de la cadena *format*.
`-f`	Usa saltos de página en lugar de retornos de carro para separar páginas.
`-h "`*`header`*`"`	En la porción central del encabezado de página, usa *header* en lugar del nombre del archivo que está siendo procesado.
`-l `*`lenght`*	Establece la longitud de página a *length*. Por defecto es 66 (Tamaño US letter a 6 líneas por pulgada)
`-n`	Numera las líneas.
`-o `*`offset`*	Crea un margen izquierdo de *offset* caracteres de ancho.

`-w` *`width`*	Establece el ancho de página a *witdth*. Por defecto es 72.

`pr` se usa a menudo como filtro en entubados. En este ejemplo, produciremos un listado del directorio `/usr/bin` y los formatearemos paginado en 3 columnas usando `pr`:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -w 65 | head



2009-02-18 14:00 Page 1
[ apturl bsd-write
411toppm ar bsh
a2p arecord btcflash
a2ps arecordmidi bug-buddy

a2ps-lpr-wrapper ark buildhash
```

Enviando un trabajo de impresión a una impresora

La suite de impresión CUPS soporta dos métodos para imprimir usados históricamente en sistemas como-Unix. Un método, llamado Berkeley o LPD (usado en la versión de Unix Berkeley Software Distribution), usa el programa `lpr`, mientras que el otro método, llamado SysV (De la versión de Unix System V), usa el programa `lp`. Ambos programas hacen básicamente lo mismo. Usar uno u otro es cuestión de gustos.

lpr - Imprime archivos (Estilo Berkeley)

El programa `lpr` puede usarse para enviar archivos a la impresora. También puede usarse en entubados, ya que acepta entrada estándar. Por ejemplo, para imprimir resultados de nuestro directorio multicolumna listado antes, podríamos hacer ésto:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 | lpr
```

y el informe será enviado a la impresora por defecto del sistema. Para enviar el archivo a una impresora distinta, puede usarse la opción `-P` así:

```
lpr -P nombre_impresora
```

donde *nombre_impresora* es el nombre de la impresora que queremos. Para ver una lista de impresoras conocidas por el sistema:

```
[me@linuxbox ~]$ lpstat -a
```

Consejo: Muchas distribuciones Linux permiten definir una "impresora" que produce archivos en PDF (Portable Document Format - Formato de Documento Portable), en lugar de imprimirla en la impresora física. Esto es muy útil para experimentar con comandos de impresión. Comprueba tu programa de configuración de impresora para ver si soporta esta configuración. En algunas distribuciones, puedes necesitar instalar paquetes adicionales (como `cups-pdf`) para habilitar esta capacidad.

Aquí tienes algunas de las opciones comunes de `lpr`:

Tabla 22-2: Opciones comunes de lpr

Opción	Descripción
`-# número`	Establece el número de copias a *número*.
`-p`	Imprime cada página con una cabecera sombreada con la fecha, hora, nombre del trabajo y número de página. Esta opción denominada "impresión bonita" puede usarse cunado imprimimos archivos de texto.
`-P impresora`	Especifica el nombre de la impresora usada para la salida. Si no se especifica una impresora, se usa la impresora por defecto del sistema.
`-r`	Borra archivos tras imprimirlos. Esto sería útil para programas que producen archivos temporales de impresora.

lp - Imprime archivos (Estilo System V)

Como `lpr`, `lp` acepta tanto archivos como entrada estándar para imprimir. Se diferencia de `lpr` en que soporta una colección de opciones diferente (y algo más sofisticada). Aquí tenemos las opciones comunes:

Tabla 22-3: Opciones comunes de lp

Opción	Descripción
`-d impresora`	Establece el destino (impresora) a *impresora*. Si no se especifica la opción d, se usa la impresora por defecto del sistema.
`-n número`	Establece el número de copias a *número*.
`-o landscape`	Establece la salida a orientación apaisada.
`-o fitplot`	Escala el archivo para encajarlo a la página. Esto es útil cuando imprimimos imágenes, como archivos JPEG.
`-o scaling=número`	Escala el archivo a *número*. El valor 100 rellena la página. Los valores menores de 100 se reducen, mientras que los valores mayores de 100 hacen que esl archivo se imprima en varias páginas.
`-o cpi=número`	Establece los caracteres por pulgada de salida a *número*. La opción por defecto es 10.
`-o lpi=número`	Establece las líneas por pulgada de salida a *número*. La opción por defecto es 6.

`-o page-bottom=puntos`
`-o page-left=puntos`
`-o page-right=puntos`
`-o page-top=puntos`
	Establece los márgenes de página. Los valores se expresan en *puntos*, una unidad de medida tipográfica. Hay 72 puntos en una pulgada.

`-P páginas`	Especifica la lista de páginas. Las *páginas* pueden

expresarse como una lista separada por comas y/o un rango. Por ejemplo "1,3,5,7-10"

Generaremos nuestro listado de directorio de nuevo, esta vez imprimiendo 12 CPI y 8 LPI con un margen izquierdo de media pulgada. Fijate que hemos ajustado las opciones de pr para que cuente para el nuevo tamaño de página:

```
[me@linuxbox ~]$ ls /usr/bin | pr -4 -w 90 -l 88 | lp -o page-
left=36 -o cpi=12 -o lpi=8
```

Este entubado produce un listado de cuatro columnas usando un tamaño de letra más pequeño que el que viene por defecto. El incremento de número de caracteres por pulgada nos permite ajustar más columnas en la página.

Otra opción: a2ps

El programa a2ps es interesante. Como podemos suponer por su nombre, es un programa de conversión de formatos, pero también es algo más. Su nombre originalmente significa "ASCII to PostScript - ASCII a PostScript" y se usaba para preparar archivos de texto para imprimir en impresoras PostScript. A lo largo de los años, sin embargo, las capacidades del programa han crecido, y ahora su nombre significa "Cualquier cosa a PostScript". Mientras que su nombre nos sugiere un programa de conversión de formatos, es realmente un programa de impresión. Manda su salida por defecto a la impresora predeterminada del sistema en lugar de a la salida estándar. El comportamiento del programa por defecto es como el de una "impresora bonita", lo que significa que mejora la apariencia de la salida. Si usamos el programa para crear un archivo PostScript en nuestro escritorio:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -t | a2ps -o ~/Desktop/ls.ps
-L 66
[stdin (plain): 11 pages on 6 sheets]
[Total: 11 pages on 6 sheets] saved into the file
`/home/me/Desktop/ls.ps'
```

Aquí filtramos la cadena con pr, usando la opción -t (que omite encabezados y pies) y luego con a2ps, especificamos un archivo de salida (opción -o) y 66 líneas por página (opción -L) para igualar la paginación de pr. Si vemos el archivo resultante con un visor de archivos adecuado, veremos esto:

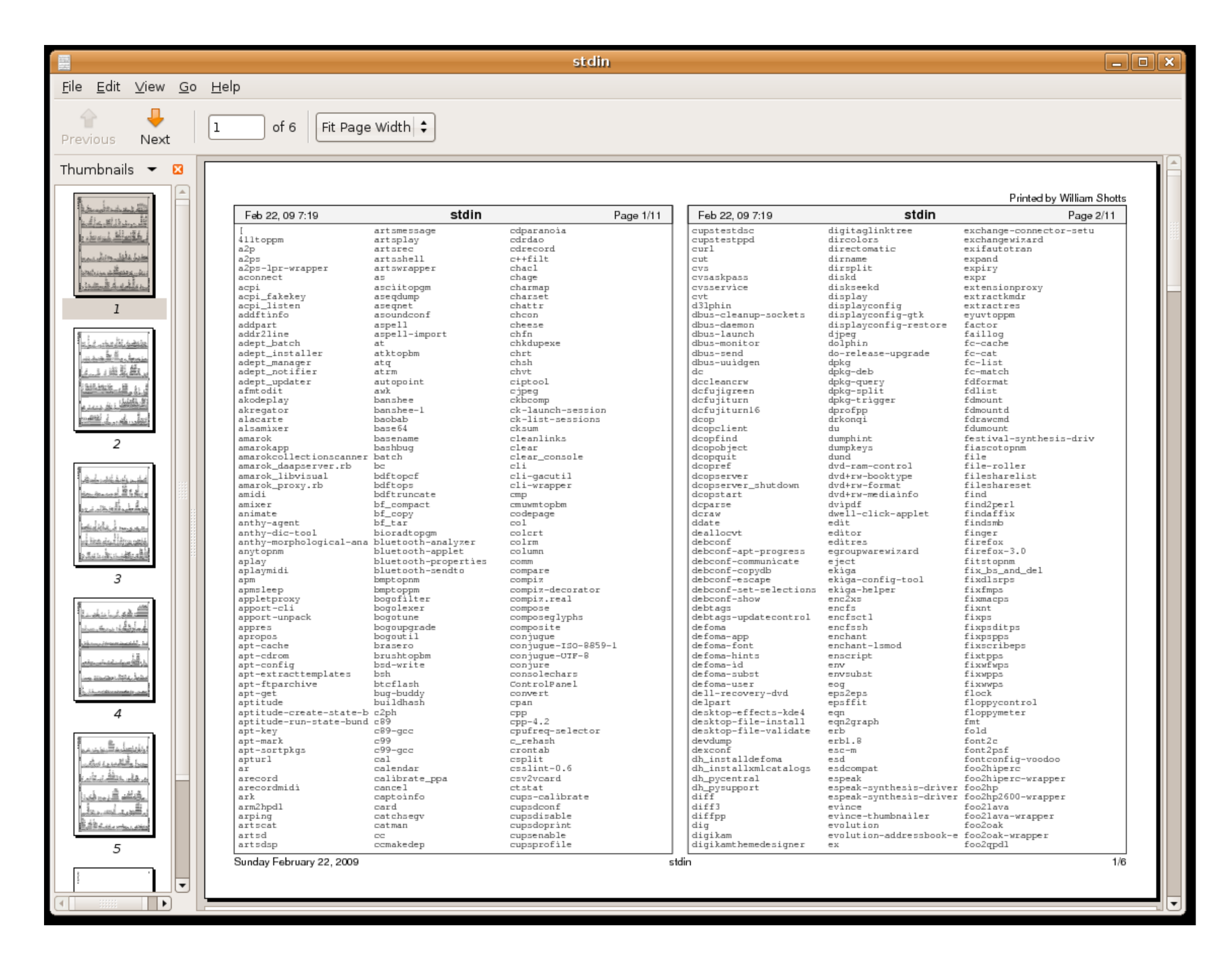


Figura 6: Viendo la salida de a2ps

Como podemos ver, la plantilla de salida por defecto es en formato dos páginas por hoja. Esto provoca que el contenido de dos pagínas se impriman en cada hoja de papel. `a2ps` aplica bonitos encabezados y pies de página también.

`a2ps` tiene muchas opciones. Aquí tenemos un sumario:

Tabla 22-4: Opciones de a2ps

Opción	**Descripción**
`--center-tittle` *`texto`*	Establece el título del centro de la página a *texto*.
`--columns` *`número`*	Ordena páginas en *número* columnas. La opción por defecto 2.
`--footer` *`texto`*	Establece el pie de página a *texto*.
`--guess`	Muestra los tipo de archivos dados como argumentos. Como `a2ps` trata de convertir todos los tipos de datos, esta opción puede ser útil para predecir que hará `a2ps` cuando se le de un archivo en particular.
`--left-footer` *`texto`*	Establece el pie izquierdo de la página a *texto*.

`--left-title` *texto*	Establece el título izquierdo de la página a *texto*.
`--line-numbers=`*intervalo*	Numera las líneas de salida cada *intervalo* líneas.
`--list=defaults`	Muestra la configuración por defecto.
`--list=`*tema*	Muestra las configuración de *tema*, donde *tema* es uno de los siguientes: delegaciones (programas externos que se usarán para convertir datos), codificación, características, variables, medios (tamaño de papel y similares), ppd (descripciones de impresora PostScript), impresoras, prólogos (porciones de código que aparecen antes de la salida normal), hojas de estilo y opciones de usuario.
`--pages` *rango*	Imprime páginas en *rango*.
`--right-footer` *texto*	Establece el pie derecho de la página a *texto*.
`--right-title` *texto*	Establece el título derecho de la página a *texto*.
`--rows` *número*	Ordena páginas en *número* filas. Por defecto es una.
`-B`	Sin encabezados de página.
`-b` *texto*	Establece el encabezado de página a *texto*.
`-f` *tamaño*	Usa fuente de *tamaño* puntos.
`-l` *número*	Establece los caracteres por línea a *número*. Ésta y la opción `-L` (a continuación) pueden usarse para hacer que archivos paginados con otros programas, como `pr`, encajen correctamente en la página.
`-L` *número*	Establece las líneas por página a *número*.
`-M` *nombre*	Usa el medio *nombre*. Por ejemplo, "A4"
`-n` *número*	Muestra el *número* de copias de cada página.
`-o` *archivo*	Envía la salida a archivo. Si archivo se especifica como "-", usa la salida estándar.
`-P` *impresora*	Usa *impresora*. Si no se especifica la impresora, se usa la impresora por defecto del sistema.
`-R`	Orientación retrato.
`-r`	Orientación apaisada.
`-T` *número*	Establece etiquetas de stop para cada *número* caracteres.
`-u` *texto*	Sobrescribe (marca de agua) páginas con *texto*.

Esto es sólo un resumen. `a2ps` tiene varias opciones más.

Nota: `a2ps` está todavía en desarrollo activo. Durante mi prueba, he notado comportamientos diferentes en varias distribuciones. En CentOS 4, la salida siempre va a la salida estándar por defecto. En CentOS 4 y Fedora 10, la salida va por defecto a un medio A4, a pesar de que el programa esté configurado para usar un medio tamaño carta por defecto. Podría superar estos problemas especificando específicamente la opción deseada. En Ubuntu 8.04, `a2ps` se comporta como está documentado.

Ten en cuenta también que hay otro formateador de salida útil para convertir texto a PostScript. Se llama `enscript`, puede realizar muchos de los mismos tipos de trucos de formateo e impresión, pero al contrario que `a2ps`, sólo acepta entrada de texto.

Monitorizando y controlando trabajos de impresión

Como los sistemas de impresión Unix están diseñados para manejar múltiples trabajos de impresión de múltiples usuarios, CUPS está diseñado para hacer lo mismo. A cada impresora se le da una *cola de impresión*, donde se aparcan los trabajos hasta que pueden ser *puestos en cola* hacia la impresora. CUPS proporciona varios programas de línea de comandos que se usan para manejar estados y colas de impresión. Como los programas `lpr` y `lp`, estos programas de gestión siguen el modelo de los sistemas de impresión Berkeley y System V.

lpstat - Muestra el estado del sistema de impresión

El programa `lpstat` es útil para determinar los nombres y la disponibilidad de las impresoras en el sistema. Por ejemplo, si tuviéramos un sistema con una impresora física (llamada "printer") y una impresora virtual (llamada "PDF"), podríamos comprobar su estado así:

```
[me@linuxbox ~]$ lpstat -a
PDF accepting requests since Mon 08 Dec 2008 03:05:59 PM EST
printer accepting requests since Tue 24 Feb 2009 08:43:22 AM EST
```

Además, podríamos determinar una descripción más detallada de la configuración del sistema de impresión de esta forma:

```
[me@linuxbox ~]$ lpstat -s
system default destination: printer
device for PDF: cups-pdf:/
device for printer: ipp://print-server:631/printers/printer
```

En este ejemplo, vemos que "printer" es la impresora por defecto del sistema y que es una impresora de red que usa el Protocolo de Impresión de Internet - Internet Printing Protocol (ipp://) que está conectada a un sistema llamado "print-server".

Las opciones útiles usada más habitualmente incluyen:

Tabla 22-5: Opciones comunes de lpstat

Opción	Descripción
`-a [`*`impresora`*`...]`	Muestra el estado de la cola de impresión de *impresora*. Fíjate que es el estado de la capacidad de la cola de la impresora de aceptar trabajos, no el estado de las impresoras físicas. Si no se especifican impresoras, se muestran todas las colas de impresión.
`-d`	Muestra el nombre de la impresora por defecto del sistema.

`-p [`*`impresora`*`...]`	Muestra el estado de la *impresora* especificada. Si no se especifican impresoras, se muestran todas.
`-r`	Muestra el estado de los servidores de impresión.
`-s`	Muestra un sumario de estado.
`-t`	Muestra un completo informe de estado.

lpq - Muestra el estado de la cola de impresión

Para ver el estado de una cola de impresión, se utiliza el programa `lpq`. Esto nos permite ver el estado de la cola y de los trabajos de impresión que contiene. Aquí tenemos un ejemplo de una cola vacía de una impresora por defecto del sistema llamada "printer":

```
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

Si no especificamos una impresora (usando la opción `-P`), se muestra la impresora por defecto del sistema. Si enviamos un trabajo a la impresora y luego vemos la cola, lo veremos listado:

```
[me@linuxbox ~]$ ls *.txt | pr -3 | lp
request id is printer-603 (1 file(s))
[me@linuxbox ~]$ lpq
printer is ready and printing
Rank Owner Job File(s) Total Size

active me 603 (stdin) 1024 bytes
```

lprm/cancel - Cancela trabajos de impresión

CUPS aporta dos programas usados para terminar trabajos de impresión y eliminarlos de la cola de impresión. Uno es estilo Berkeley (`lprm`) y el otro es System V (`cancel`). Difieren ligéramente en la opciónes que soportan, pero basícamente hacen lo mismo. Usando nuestro trabajo de impresión anterior como ejemplo, podríamos parar el trabajo y eliminarlo de la siguiente forma:

```
[me@linuxbox ~]$ cancel 603
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

Cada comando tiene opciones para eliminar todos los trabajos pertenecientes a un usuario en particular, una impresora en particular y múltiples números de trabajos. Sus respectivas man pages tienen todos los detalles.

Resumiendo

En este capítulo, hemos visto como las impresoras del pasado han influido el diseño de los sistema de impresión de las máquinas como-Linux, y cuanto control tenemos en la línea de comandos para controlar no sólo la programación y ejecución de trabajos de impresión, si no también la variedad de opciones de salida.

Para saber más

- Un buen artículo sobre el lenguaje de descripción de página PostScript:

http://en.wikipedia.org/wiki/PostScript

- El Common Unix Printing System - Sistema de impresión común de Unix (CUPS):

http://en.wikipedia.org/wiki/Common_Unix_Printing_System
http://www.cups.org/

- Los sistemas Berkeley y System V:

http://en.wikipedia.org/wiki/Berkeley_printing_system

http://en.wikipedia.org/wiki/System_V_printing_system

Compilando programas

En este capítulo, veremos cómo construir programas compilando código fuente. La disponibilidad de código fuente es la libertad fundamental que hace que Linux sea posible. El ecosistema completo de desarrollo Linux se basa en el libre intercambio entre desarrolladores. Para muchos usuarios de escritorio, compilar es un arte perdido. Solía ser bastante común, pero hoy, los proveedores e distribuciones mantienen amplios repositorios de binarios precompilados, listos para descargar y usar. En el momento de la escritura de este libro, el repositorio Debian (uno de los más grandes de todas las distribuciones) contiene casi 23.000 paquetes.

Entonces ¿por qué compilar software? Hay dos razones:

1. **Disponibilidad**. A pesar del número de programas precompilados en los repositorios de las distribuciones, alguno distribuidores pueden no incluir todas las aplicaciones deseadas. En este caso, la única forma de obtener el programa deseado es compilarlo de su fuente.
2. **Oportunidad**. Mientras que algunos distribuciones se especializan en las últimas versiones de los programas, muchas no lo hacen. Esto significa que para tener la última versión de un programa es necesario compilarlo.

Compilar software desde código fuente puede llegar a ser muy complejo y técnico; mucho más allá del alcance de muchos usuarios. Sin embargo, muchas tareas de compilación son bastante fáciles y sólo necesitan unos pocos pasos. Todo depende del paquete. Veremos un caso muy simple para hacer un repaso del proceso y como punto de partida para aquellos que quieran emprender un estudio más a fondo.

Presentaremos un nuevo comando:

- `make` - Utilidad para mantener programas

¿Qué es compilar?

Digamos que, compilar es el proceso de traducir *código fuente* (la descripción legible por humanos de un programa escrito por un programador) al lenguaje nativo del procesador del ordenador.

El procesador del ordenador (o *CPU*) trabaja a un nivel muy elemental, ejecutando programas en lo que se llama *lenguaje máquina*. Es un código numérico que describe operaciones muy pequeñas, como "añade este byte", "apunta a esta localización en la memoria" o "copia este byte."

Cada una de esas instrucciones se expresan en binario (unos y ceros). Los primeros programas de ordenador se escribieron usando este código numérico, que explicaría porque los que los escribieron

se dice que fumaban mucho, bebían litros de café y usaban gafas con cristales gordos.

El problema se solucionó con la llegada del *lenguaje ensamblador*, que reemplazaba los códigos numéricos con caracteres *mnemotécnicos* (ligeramente) más fáciles de usar como CPY (para copiar) y MOV (para mover). Los programas escritos en lenguaje ensamblador se procesan a lenguaje máquina por un programa llamado *ensamblador*. El lenguaje ensamblador se usa aún hoy para ciertas tareas especializadas de programación, como *controladores de dispositivos* y *sistemas embebidos*.

A continuación llegamos a lo que se llama *lenguajes de programación de alto nivel*. Se llaman así porque permiten que el programador esté menos preocupado con los detalles de lo que está haciendo el procesador y más con resolver los problemas que tiene entre manos. Los primeros (desarrollados durante los años cincuenta) incluyen FORTRAN (diseñado para tareas científicas y técnicas) y COBOL (diseñado para aplicaciones comerciales). Ambos tiene un uso limitado todavía hoy.

Aunque hay muchos lenguajes de programación populares, dos predominan. La mayoría de programas escritos para sistemas modernos están escritos en *C* o *C++*. En los ejemplos que siguen, compilaremos un programa en C.

Los programas escritos en lenguajes de programación de alto nivel son convertidos a lenguaje máquina procesándolos en otro programa, llamado *compilador*. Algunos compiladores traducen las instrucciones de alto nivel en lenguaje ensamblador y luego usan un ensamblador para realizar el último paso de traducirlo a lenguaje máquina.

Un proceso usado a menudo junto con el compilado es el llamado *enlazado*. Hay muchas tareas comunes realizadas por programas. Tomemos, por ejemplo, abrir un archivo. Muchos programas realizan esta tarea, pero sería un despilfarro que cada programa implemente su propia rutina para abrir archivos. Tiene más sentido tener una única pieza de programación que sepa como abrir archivos y permitir a todos los programas que lo necesiten compartirla. Dar soporte a tareas comunes es realizado por las llamadas *librerías*. Contienen múltiples *rutinas*, cada una realiza alguna tarea común que puede ser compartida por múltiples programas. Si miramos en los directorios `/lib` y `/usr/lib`, podemos ver donde están muchas de ellas. Un programa llamado enlazador se usa para realizar las conexiones entre la salida del compilador y las librerías que el programa compilado requiere. El resultado final de este proceso es el *archivo ejecutable del programa*, listo para ser usado.

¿Todos los programas se compilan?

No. Como hemos visto, hay programas como los scripts de shell que no requieren compilación. Se ejecutan directamente. Están escritos en lo que se conoce como *lenguajes de script o interpretados*. Estos lenguajes han ganado popularidad en los últimos años e incluyen *Perl, Python, PHP, Ruby* y muchos otros.

Los lenguajes de script se ejecutan por un programa especial llamado *intérprete*. Un intérprete toma el archivo del programa y lee y ejecuta cada instrucción contenida dentro de él. En general, los programas interpretados se ejecutan mucho más lentamente que los programas compilados. Ésto es porque cada instrucción de código en un programa interpretado se traduce cada vez que se ejecuta, mientras que en un programa compilado, una instrucción de código fuente se traduce sólo una vez, y esta traducción se graba permanentemente en el archivo ejecutable final.

Entonces ¿por qué son tan populares los programas interpretados? Para muchas tareas de programación, los resultados son "suficientemente rápidos", pero la ventaja real es que es generalmente más rápido y fácil desarrollar programas interpretados que programas compilados. Los programas se desarrollan a menudo en un ciclo repetitivo de código, compilación, prueba. A medida que un programa crece en tamaño, la fase de compilación del ciclo puede llegar a ser muy larga. Los lenguajes interpretados eliminan el paso de la compilación y por lo tanto aumentan la velocidad de desarrollo del programa.

Compilando un programa en C

Compilemos algo. Antes de hacerlo sin embargo, vamos a necesitar algunas herramientas como el compilador, el enlazador y `make`. El compilador C usado casi universalmente en el entorno Linux se llama `gcc` (GNU C Compiler - Compilador C GNU), originalmente escrito por Richard Stallman. La mayoría de las distribuciones no instalan gcc por defecto. Podemos comprobar si el compilador esta presente así:

```
[me@linuxbox ~]$ which gcc
/usr/bin/gcc
```

El resultado de este ejemplo indica que el compilador está instalado.

Consejo: Tu distribución puede tener un meta-paquete (una colección de paquetes) para el desarrollo de software. Si es así, considera instalarlo si quieres compilar programas en tu sistema. Si tu sistema no ofrece un meta-paquete, prueba a instalar los paquetes `gcc` y `make`. En muchas distribuciones, es suficiente para realizar el ejercicio siguiente.

Obteniendo el código fuente

Para nuestro ejercicio de compilación, vamos a compilar un programa del Proyecto GNU llamado `diction`. Es un pequeño pero práctico programa que comprueba la calidad de escritura y edición de los archivos de texto. Como programa, es bastante pequeño y fácil de construir.

Siguiendo la norma, primero vamos a crear un directorio para nuestro código fuente llamado `src` y luego descargaremos el código fuente en él usando `ftp`:

```
[me@linuxbox ~]$ mkdir src
[me@linuxbox ~]$ cd src
[me@linuxbox src]$ ftp ftp.gnu.org
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:me): anonymous
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd gnu/diction
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r--r-- 1 1003 65534 68940 Aug 28 1998 diction-0.7.tar.gz
-rw-r--r-- 1 1003 65534 90957 Mar 04 2002 diction-1.02.tar.gz
```

```
-rw-r--r-- 1 1003 65534 141062 Sep 17 2007 diction-1.11.tar.gz
226 Directory send OK.
ftp> get diction-1.11.tar.gz
local: diction-1.11.tar.gz remote: diction-1.11.tar.gz
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for diction-1.11.tar.gz
(141062 bytes).
226 File send OK.
141062 bytes received in 0.16 secs (847.4 kB/s)
ftp> bye
221 Goodbye.
[me@linuxbox src]$ ls
diction-1.11.tar.gz
```

Nota: Como somos los "mantenedores" de este código fuente mientras lo compilamos, lo guardaremos en `~/src`. El código fuente instalado por nuestra distribución se instalará en `/usr/src`, mientras que nuestro código fuente destinado al uso de múltiples usuarios se instala a menudo en `/usr/local/src`.

Como podemos ver, el código fuente se proporciona normalmente en forma de archivo tar comprimido. Algunas veces se llama *tarball*, este archivo contiene el *source tree* (árbol de código), o jerarquía de directorios y archivos que abarca el código fuente. Tras llegar al sitio ftp, examinamos la lista de archivos tar disponibles y seleccionamos la versión más nueva para descargar. Usando el comando `get` contenido en `ftp`, copiamos el archivo del servidor ftp a la máquina local.

Una vez que se ha descargado el archivo tar, tiene que ser desempaquetado. Esto se hace con el programa `tar`:

```
[me@linuxbox src]$ tar xzf diction-1.11.tar.gz
[me@linuxbox src]$ ls
diction-1.11 diction-1.11.tar.gz
```

Consejo: El programa `diction`, como todo el software del Proyecto GNU, sigue ciertos estándares para el empaquetado de código fuente. La mayoría del resto de código fuente disponible en el ecosistema Linux también sigue este estándar. Un elemento del estándar es que cuando el código fuente del archivo tar es desempaquetado, se creará un directorio que contiene el árbol fuente, y que este directorio se llamará *project-x.xx*, conteniendo así tanto el nombre del proyecto como el número de versión. Este esquema también permite la instalación fácil de múltiples versiones del mismo programa. Sin embargo, a menudo es una buena idea examinar la disposición del árbol antes de desempaquetarlo. Algunos proyectos no crearán el directorio, pero en su lugar colocarán los archivos directamente en el directorio actual. Esto provocará un desorden en tu, de otra forma, bien organizado directorio `src`. Para evitar esto, usa el siguiente comando para examinar el contenido del archivo tar:

```
tar tzvf archivotar | head
```

Examinando el árbol fuente

Desempaquetar el archivo tar da como resultado la creación de un nuevo directorio, llamado `diction-1.11`. Este directorio contiene el árbol fuente. Miremos dentro:

```
[me@linuxbox src]$ cd diction-1.11
[me@linuxbox diction-1.11]$ ls
config.guess diction.c getopt.c nl
config.h.in diction.pot getopt.h nl.po
config.sub diction.spec getopt_int.h README
configure diction.spec.in INSTALL sentence.c
configure.in diction.texi.in install-sh sentence.h
COPYING en Makefile.in style.1.in
de en_GB misc.c style.c
de.po en_GB.po misc.h test
diction.1.in getopt1.c NEWS
```

En él, vemos un número de archivos. Los programa pertenecientes al Proyecto GNU, así como muchos otros, proporcionarán los archivos de documentación `README`, `INSTALL`, `NEWS` y `COPYNG`. Estos archivos contienen la descripción del programa, información de cómo construirlo e instalarlo, y sus términos de licencia. Es siempre una buena idea leer los archivos `README` e `INSTALL` antes de intentar construir el programa.

Los otros archivos interesantes en este directorio son los que terminan en `.c` y `.h`:

```
[me@linuxbox diction-1.11]$ ls *.c
diction.c getopt1.c getopt.c misc.c sentence.c style.c
[me@linuxbox diction-1.11]$ ls *.h
getopt.h getopt_int.h misc.h sentence.h
```

Los archivos `.c` contienen los dos programas C proporcionados por el paquete (`style` y `diction`), divididos en módulos. Es una práctica común para programas grandes dividirlos en trozos más pequeños y fáciles de manejar. Los archivos de código fuente son de texto ordinario y pueden examinarse con `less`:

```
[me@linuxbox diction-1.11]$ less diction.c
```

Los archivos `.h` se conocen como *archivos de cabecera (header files)*. Estos, también, son texto ordinario. Los archivos cabecera contienen descripciones de las rutinas incluidas en un archivo de código fuente o librería. Para que el compilador pueda conectar los módulos, debe recibir una descripción de los módulos necesarios para completar todo el programa. Por el principio del archivo `diction.c`, vemos esta línea:

```
#include "getopt.h"
```

Ésto ordena al compilador a leer el archivo `getopt.h` como si leyera el código fuente de `diction.c` para poder "saber" que hay en `getopt.c.` El archivo `getopt.c` proporciona rutinas que son compartidas por los programas `style` y `diction`.

Encima de la declaración `include` de `getopt.h`, vemos otras declaraciones `include` como estas:

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <unistd.h>
```

Estas se refieren también a archivos cabecera, pero se refieren a archivos cabecera que se encuentran fuera del árbol de código actual. Son proporcionados por el sistema para soportar la compilación de cada programa. Si miramos en `/usr/include`, podemos verlos:

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

Los archivos cabecera en este directorio se instalaron cuando instalamos el compilador.

Construyendo el programa

La mayoría de los programas se construyen con una secuencia simple de dos comandos:

```
./configure
make
```

El programa `configure` es un script de shell que es proporcionado por el árbol fuente. Su trabajo es analizar el *entorno de construcción*. La mayoría del código fuente se diseña par ser *portable*. O sea, se diseña para construirse en más de un tipo de sistema como-Unix. Pero para hacer eso, el código fuente puede necesitar someterse a leves ajustes durante la construcción para acomodarse a las diferencias entre sistemas. `configure` también comprueba qué herramientas externas necesarias y componentes están instalados. Ejecutemos `configure`. Como `configure` no está localizado donde el shell espera normalmente que estén almacenados los programas, tenemos que decirle al shell explícitamente su localización precediendo el comando con ./ para indicar que el programa se localiza en el directorio de trabajo actual:

```
[me@linuxbox diction-1.11]$ ./configure
```

`configure` producirá un montón de mensajes a medida que prueba y configura la construcción. Cuando termina, tendrá un aspecto como éste:

```
checking libintl.h presence... yes
checking for libintl.h... yes
checking for library containing gettext... none required
configure: creating ./config.status
config.status: creating Makefile
config.status: creating diction.1
config.status: creating diction.texi
config.status: creating diction.spec
config.status: creating style.1
config.status: creating test/rundiction
config.status: creating config.h
[me@linuxbox diction-1.11]$
```

Lo importante aquí es que no hay mensajes de error. Si los hubiera, la configuración fallaría, y el programa no se construiría hasta que se corrigieran los errores.

Vemos que `configure` ha creado varios archivos nuevos en nuestro directorio fuente. El más importante es `Makefile`. `Makefile` es un archivo de configuración que indica al programa `make` como construir exactamente el programa. Sin él, `make` no funcionará. `Makefile` es un

archivo de texto ordinario, así que podemos verlo:

```
[me@linuxbox diction-1.11]$ less Makefile
```

El programa `make` toma como entrada un *makefile* (que normalmente se llama `Makefile`), que describe las relaciones y dependencias entre los componentes que componen el programa finalizado.
La primera parte de makefile define variables que son sustituidas en secciones posteriores del makefile. Por ejemplo vemos la línea:

```
CC= gcc
```

que define que el compilador C será `gcc`. Más adelante en el makefile, vemos una instancia donde se usa:

```
diction: diction.o sentence.o misc.o getopt.o getopt1.o
$(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \
getopt.o getopt1.o $(LIBS)
```

Aquí se realiza una sustitución, y el valor `$(CC)` se reemplaza por `gcc` en el momento de la ejecución.

La mayoría del makefile consiste en dos líneas, que definen un *objetivo*, en este caso el archivo ejecutable `diction`, y los archivos de los que depende. Las líneas restantes describen lo que el/los comando/s necesitan para crear el objetivo desde sus componentes. Vemos en este ejemplo que el archivo ejecutable `diction` (uno de los productos finales) depende de la existencia de `diction.o`, `sentence.o`, `misc.o`, `getop.o` y `gestopt1.o`. Mas adelante aún, en el makefile, vemos las definiciones de cada uno de estos objetivos:

```
diction.o: diction.c config.h getopt.h misc.h sentence.h
getopt.o: getopt.c getopt.h getopt_int.h
getopt1.o: getopt1.c getopt.h getopt_int.h
misc.o: misc.c config.h misc.h
sentence.o: sentence.c config.h misc.h sentence.h
style.o: style.c config.h getopt.h misc.h sentence.h
```

Sin embargo, no vemos ningún comando especificado para ellos. Esto es gestionado por un objetivo general, anteriormente en el archivo, que describe el comando usado para compilar cualquier archivo `.c` en un archivo `.o`:

```
.c.o:
$(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

Todo esto parece muy complicado. ¿Por qué no listamos simplemente todos los pasos para compilar las partes y terminamos? La respuesta a ésto se aclarará en un momento. Mientras tanto, ejecutemos `make` y construyamos nuestros programas:

```
[me@linuxbox diction-1.11]$ make
```

El programa `make` se ejecutará, usando los contenidos de `Makefile` para guiar sus acciones. Producirá un montón de mensajes.

Cuando termine, veremos que todos los objetivos están presentes ahora en nuestro directorio:

```
[me@linuxbox diction-1.11]$ ls
config.guess de.po en install-sh s entence.c
config.h diction en_GB Makefile sentence.h
config.h.in diction.1 en_GB.mo Makefile.in sentence.o
config.log diction.1.in en_GB.po misc.c style
config.status diction.c getopt1.c misc.h style.1
config.sub diction.o getopt1.o misc.o style.1.in
configure diction.pot getopt.c NEWS style.c
configure.in diction.spec getopt.h nl style.o
COPYING diction.spec.in getopt_int.h nl.mo test
de diction.texi getopt.o nl.po
de.mo diction.texi.in INSTALL README
```

Entre los archivos, vemos `diction` y `style`, los programas que elegimos construir. ¡Felicidades están en orden! ¡Acabamos de compilar nuestros primeros programas desde código fuente!

Pero sólo por curiosidad, ejecutemos `make` de nuevo:

```
[me@linuxbox diction-1.11]$ make
make: Nothing to be done for `all'.
```

Sólo produce un extraño mensaje. ¿Qué está pasando? ¿Por qué no ha construido el programa de nuevo? Ah, esta es la magia de `make`. En lugar de simplemente construirlo todo de nuevo, `make` sólo construye lo que necesita construirse. Con todos los objetivos presentes, `make` ha determinado que no hay nada que hacer. Podemos demostrar ésto eliminando uno de los objetivos y ejecutando make de nuevo para ver qué hace. Deshagámonos de uno de los objetivos intermedios:

```
[me@linuxbox diction-1.11]$ rm getopt.o
[me@linuxbox diction-1.11]$ make
```

Vemos que `make` reconstruye y reenlaza los programas `diction` y `style`, ya que dependen del módulo perdido. Este comportamiento también indica otra característica importante de `make`: mantiene los objetivos actualizados. `make` insiste en que los objetivos sean más nuevos que sus dependencias. Esto tiene todo el sentido, como programador a menudo actualizaras algo de código fuente y luego usarás `make` para construir una nueva versión del producto finalizado. `make` se asegura de que todo los que necesite construirse en el código actualizado sea construido. Si usamos el programa `touch` para "actualizar" uno de los archivos de código fuente, podemos ver lo que ocurre:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me me 37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me me 33125 2007-03-30 17:45 getopt.c
[me@linuxbox diction-1.11]$ touch getopt.c
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me me 37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me me 33125 2009-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

Después de que `make` se ejecute, vemos que ha restaurado el objetivo para que sea más nuevo que la dependencia:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me me 37164 2009-03-05 06:24 diction
-rw-r--r-- 1 me me 33125 2009-03-05 06:23 getopt.c
```

La capacidad de `make` de construir inteligentemente sólo lo que necesita ser construido es un gran beneficio para los programadores. Aunque los ahorros de tiempo no son muy aparentes en nuestro pequeño proyecto, es muy significante para proyectos más grandes. Recuerda, el kernel Linux (un programa sometido a modificaciones y mejoras constantes) contiene varios *millones* de líneas de código.

Instalando el programa

El código fuente bien empaquetado ofrece a menudo un objetivo especial de `make` llamado `install`. Este objetivo instalará el producto final en un directorio de sistema para su uso. Usualmente, este directorio es `/usr/local/bin`, la localización tradicional para construir software localmente. Sin embargo, este directorio no es normalmente modificable por los usuarios normales, así que tenemos que ser superusuario para realizar la instalación:

```
[me@linuxbox diction-1.11]$ sudo make install
```

Después de realizar la instalación, podemos comprobar que el programa está listo:

```
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
```

¡Y ahí lo tenemos!

Resumiendo

En este capítulo, hemos visto como tres simples comandos:

```
./configure
make
make install
```

pueden usarse para construir muchos paquetes de código fuente. También hemos visto el importante rol que `make` juega en el mantenimiento de programas. El programa `make` puede usarse para cualquier tarea que se necesite para mantener una relación objetivo/dependencia, no sólo para compilar código fuente.

Para saber más

- La wikipedia tiene buenos artículos sobre compiladores y el programa `make`:

http://en.wikipedia.org/wiki/Compiler
http://en.wikipedia.org/wiki/Make_(software)

- *El manual de GNU make*:

http://www.gnu.org/software/make/manual/html_node/index.html

Escribiendo tu primer script

En los capítulos precedentes, hemos reunido un arsenal de herramientas de línea de comandos. Aunque estas herramientas pueden resolver muchos tipos de problemas de computación, todavía estamos limitados a usarlas manualmente una a una en la línea de comandos. ¿No sería genial si pudiéramos usar el shell para hacer la mayoría del trabajo? Podemos. Poniendo nuestras herramientas juntas en programas diseñados por nosotros, el shell puede ejecutar secuencias completas de tareas por sí mismo. Podemos permitirle que haga esto escribiendo *scripts de shell*.

¿Que son scripts de shell?

En términos sencillos, un script de shell es un archivo que contiene una serie de comandos. El shell lee este archivo y ejecuta los comandos tal como han sido introducidos directamente en la línea de comandos.

El shell es algo único, en cuanto a que es una interfaz poderosa de línea de comandos para el sistema y un intérprete de lenguaje de script. Como veremos, la mayoría de las cosas que pueden hacerse con la línea de comandos puede hacerse con scripts, y la mayoría de las cosas que pueden hacerse con scripts pueden hacerse en la línea de comandos.

Hemos visto muchas características del shell, pero nos hemos enfocado en aquellas más usadas directamente en la línea de comandos. El shell también proporciona una serie de características que normalmente (aunque no siempre) se usan cuando escribimos programas.

Cómo escribir un script de shell

Para crear con éxito y ejecutar un script de shell, necesitamos hacer tres cosas:

1. **Escribir un script**. Los scripts de shell son normalmente archivos de texto. Así que necesitamos un editor de texto para escribirlos. Los mejores editores de texto proporcionarán *destacado sintáctico*, permitiéndonos ver una vista coloreada del código de los elementos del script. El destacado sintáctico nos ayudará a encontrar algunos tipos de errores comunes. `vim`, `gedit`, `kate` y muchos otros editores son buenos candidatos para escribir scripts.
2. **Hacer el script ejecutable**. El sistema es un poco quisquilloso en no dejar que ningún antiguo archivo de texto sea tratado como un programa, y ¡Es por una buena razón! Necesitamos dar permisos al archivo del script para que se permita la ejecución.
3. **Poner el script en algún lugar donde el shell pueda encontrarlo**. El shell automáticamente busca archivos ejecutables en ciertos directorios cuando no se especifica una ruta concreta. Para mayor comodidad, colocaremos nuestros scripts en dichos directorios.

Formato del archivo de script

Siguiendo la tradición de la programación, crearemos un programa "hola mundo" para demostrar un script tremendamente simple. Así que arranquemos nuestros editores de texto e introduzcamos el siguiente script:

```
#!/bin/bash
# This is our first script.
```

```
echo 'Hello World!'
```

La última línea de nuestro script es muy familiar, sólo un comando `echo` con una cadena de argumentos. La siguiente línea también es familiar. Parece un comentario que hemos visto usado en los archivos de configuración que hemos examinado y editado. Una cosa sobre los comentarios en scripts de shell es que también pueden aparecer al final de las líneas, así:

```
echo 'Hello World!' # This is a comment too
```

Todo lo que hay desde el símbolo # en adelante en la línea es ignorado.

Como muchas cosas, ésto funciona en la línea de comandos también:

```
[me@linuxbox ~]$ echo 'Hello World!' # This is a comment too
Hello World!
```

Aunque los comentarios son poco útiles en al línea de comandos, también funcionan.

La primera línea de nuestro script es un poco misteriosa. Parece como si fuera un comentario, ya que comienza con #, pero parece demasiado significativa para ser sólo eso. La secuencia de caracteres #! es, de hecho, una construcción especial llamada un *shebang*. El shebang se usa par decirle al sistema el nombre del intérprete que debería usarse para interpretar el script que sigue. Cada script de shell debería incluirlo en su primera línea.

Guardemos nuestro archivo de script como `hello_world`.

Permisos de ejecución

Lo siguiente que tenemos que hacer es hacer nuestro script ejecutable. Ésto se hace fácilmente usando `chmod`:

```
[me@linuxbox ~]$ ls -l hello_world
-rw-r--r-- 1 me me 63 2009-03-07 10:10 hello_world
[me@linuxbox ~]$ chmod 755 hello_world
[me@linuxbox ~]$ ls -l hello_world
-rwxr-xr-x 1 me me 63 2009-03-07 10:10 hello_world
```

Dos configuraciones comunes de permisos para los scripts; 755 para scripts que puede ejecutar todo el mundo, y 700 para scripts que sólo puede ejecutar el propietario. Fíjate que los scripts deben ser legibles para ser ejecutados.

Localización del archivo de script

Con los permisos establecidos, ahora podemos ejecutar nuestro script:

```
[me@linuxbox ~]$ ./hello_world
Hello World!
```

Para que el script funcione, debemos preceder el nombre del script con una ruta específica. Si no lo hacemos, obtenemos ésto:

```
[me@linuxbox ~]$ hello_world
bash: hello_world: command not found
```

¿Qué es ésto? ¿Qué hace que nuestro script sea diferente de otros programas? Al parecer, nada. Nuestro script está perfecto. El problema es su localización. Volviendo al capítulo 11, vimos la variable de entorno PATH y su efecto en cómo busca el sistema los programas ejecutables. Para abreviar, el sistema busca en una lista de directorios cada vez que necesita encontrar un programa ejecutable, si no se especifica una ruta concreta. Así es como el sistema sabe que tiene que ejecutar /bin/ls cuando escribimos ls en la línea de comandos. El directorio /bin es uno de los directorios donde el sistema busca automáticamente. La lista de directorios está contenida en una variable de entorno llamada PATH. La variable PATH contiene una lista separada por dos puntos de los directorios donde buscar. Podemos ver el contenido de PATH:

```
[me@linuxbox ~]$ echo $PATH
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sb
in:
/bin:/usr/games
```

Aquí vemos nuestra lista de directorios. Si nuestro script estuviera localizado en cualquiera de los directorios de la lista, nuestro problema estaría solucionado. Fíjate en el primer directorio de la lista, /home/me/bin. La mayoría de las distribuciones Linux configuran la variable PATH para contender un directorio bin en el directorio home del usuario, para permitir a los usuarios ejecutar sus propios programas. Así que si creamos el directorio bin y colocamos nuestro script en él, debería comenzar a funcionar como otros programas:

```
[me@linuxbox ~]$ mkdir bin
[me@linuxbox ~]$ mv hello_world bin
[me@linuxbox ~]$ hello_world
Hello World!
```

Y así lo hace.

Si la variable PATH no contiene el directorio, podemos añadirlo fácilmente incluyendo esta línea en nuestro archivo .bashrc:

```
export PATH=~/bin:"$PATH"
```

Después de hacer este cambio, tendrá efecto en cada nueva sesión de terminal. Para aplicar el cambio a la sesión de terminal actual, tenemos que hacer que el shell relea el archivo .bashrc. Esto puede hacerse mediante el comando source:

```
[me@linuxbox ~]$ . .bashrc
```

El comando punto (.) es un sinónimo del comando source, una función del shell que lee un archivo específico de comandos de shell y lo trata como entrada del teclado.

Nota: Ubuntu añade automáticamente el directorio ~/bin a la variable PATH si el directorio ~/bin existe cuando el archivo .bashrc del usuario se ejecute. Así que, en sistemas Ubuntu, si creamos el directorio ~/bin y luego salimos y volvemos a entrar, todo funciona.

Buenas localizaciones para los scripts

El directorio `~/bin` es un buen lugar para colocar scripts destinados a uso personal. Si escribimos un script que todo el mundo en el sistema está autorizado a usar, la localización tradicional es `/usr/local/bin`. Los scripts destinados al uso del administrador del sistema se localizan a menudo en `/usr/local/sbin`. En la mayoría de los casos, el software proporcionado localmente, ya sean scripts o programas compilados, debería localizarse en la jerarquía de `/usr/local` y no en `/bin` o `/usr/bin`. Estos directorios son especificados por el Estándar de Jerarquía del Sistema de Archivos de Linux para contener sólo archivos proporcionados y mantenidos por el distribuidor Linux.

Más trucos de formateado

Uno de los objetivos clave de una escritura seria de scripts es que sean fáciles de *mantener*; o sea, la facilidad con la que un script puede ser modificado por su autor u otros para adaptarlo a cambios necesarios. Hacer que un script sea fácil de leer y comprender es una forma de facilitar un mantenimiento sencillo.

Nombres opción largos

Muchos de los comandos que hemos estudiado cuentan tanto con nombres cortos como largos para las opciones. Por ejemplo, el comando `ls` tiene muchas opciones que pueden expresarse tanto en forma corta como larga. Por ejemplo:

```
[me@linuxbox ~]$ ls -ad
```

y:

```
[me@linuxbox ~]$ ls --all --directory
```

son comandos equivalentes. En interés de reducir la escritura, preferimos las opciones cortas cuando incluimos opciones en la línea de comandos, pero cuando escribimos scripts, las opciones largas pueden proporcionar una legibilidad mejor.

Identación y continuación de líneas

Cuando empleamos comandos largos, la legibilidad puede mejorarse separando el comando en varias líneas. En el capítulo 17, vimos un ejemplo particularmente largo del comando find:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
chmod 0600 '{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod
0700 '{}' ';' \)
```

Obviamente, este comando es un poco complicado de descifrar a primera vista. En un script, este comando podría ser más fácil de comprender si está escrito de esta forma:

```
find playground \
  \( \
    -type f \
    -not -perm 0600 \
    -exec chmod 0600 '{}' ';' \
```

```
\) \
-or \
\( \
  -type d \
  -not -perm 0700 \
  -exec chmod 0700 ‘{}’ ‘;’ \
\)
```

Usando continuaciones de línea (secuencias barra invertida-salto de línea) e indentación, la lógica de este comando complejo se describe más claramente para el lector. Esta técnica funciona en la línea de comandos, también, aunque raramente se utiliza, ya que es muy incómodo de escribir y editar. Una diferencia entre un script y la línea de comandos es que el script puede emplear caracteres de tabulación para realizar indentación, mientras que la línea de comandos no puede, ya que el tabulador se usa para activar el completado.

Configurando vim para escribir scripts

El editor de texto vim tiene muchas, muchas posibilidades de configuración. Hay varias opciones comunes que pueden facilitar la escritura de scripts:

:syntax on

activa el destacado sintáctico. Con esta configuración, diferentes elementos de la sintáxis del shell se mostrarán en colores diferentes cuando veamos un script. Esto es útil para identificar ciertos tipos de errores de programación. Se ve más chulo, también. Fíjate que para que esta característica funcione, debes tener una versión completa de `vim` instalada, y el archivo que estés editando debe tener un shebang indicando que el archivo es un script de shell. Si tienes dificultades con el comando anterior, prueba **:set syntax=sh** en su lugar.

:set hlsearch

activa la opción de destacar los resultados de búsqueda. Digamos que buscamos la palabra "echo." Con esta opción activada, cada instancia de la palabra será destacada.

`:set tabstop=4`

establece el número de columnas ocupadas por un carácter tabulador. Por defecto son 8 columnas. Estableciendo el valor a 4 (que es una práctica común) permite a las líneas largas ajustarse más fácilmente a la pantalla.

`:set autoindent`

activa la función "auto indentado". Esto hace que `vim` indente una nueva línea la misma cantidad que la línea recién escrita. Esto acelera la escritura en muchos tipos de construcción de programas. Para detener la indentación, pulsa `Ctrl-d`.

Estos cambios pueden hacerse permanentes añadiendo estos comandos (sin los dos puntos delante) a tu archivo `~/ .vimrc`.

Resumiendo

En este primer capítulo sobre scripting, hemos visto cómo se escriben los scripts y como se hace para que se ejecuten fácilmente en nuestro sistema. También vimos como podemos usar varias técnicas de formateo para mejorar la legibilidad (y por lo tanto, el mantenimiento) de nuestros scripts. En capítulos posteriores, la facilidad de mantenimiento volverá una y otra vez como un principio central en una buena escritura de scripts.

Para saber más

- Para programas "Hello World" y ejemplos en varios lenguajes de programación, ve:

http://en.wikipedia.org/wiki/Hello_world

- Este artículo de Wikipedia habla más acerca del mecanismo shebang:

http://en.wikipedia.org/wiki/Shebang_(Unix)

Comenzando un proyecto

Para comenzar con este capítulo, vamos a empezar a construir un programa. El propósito de este proyecto es ver cómo varias funciones del shell se usan para crear programas y, más importante, para crear buenos programas.

El programa que escribiremos es un *generador de informes*. Presentará varias estadísticas de nuestro sistema y su estado, y producirá este informe en formato HTML, por lo que podemos verlo en un navegador web como Firefox y Chrome.

Los programas a menudo se construyen en una serie de etapas, donde cada etapa añade funciones y capacidades. La primera etapa de nuestro programa producirá una página HTML muy minimalista que no contiene información del sistema. Eso vendrá luego.

Primera etapa: Documento minimalista

Lo primero que necesitamos saber es el formato de un documento HTML bien construido. Tiene esta pinta:

```
<HTML>
  <HEAD>
    <TITLE>Page Title</TITLE>
  </HEAD>
  <BODY>
    Page body.
  </BODY>
</HTML>
```

Si introducimos esto en nuestro editor de texto y guardamos el archivo como `foo.html`, podemos usar la siguiente URL en Firefox para ver el archivo:

```
file:///home/username/foo.html
```

La primera etapa de nuestro programa será capaz de enviar este archivo HTML a la salida estándar. Podemos escribir un programa que haga ésto muy fácilmente. Arranquemos nuestro editor de texto y creemos un nuevo archivo llamado `~/bin/sys_info_page`:

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

e introducimos el siguiente programa:

```
#!/bin/bash



# Program to output a system information page



echo "<HTML>"
echo " <HEAD>"
echo " <TITLE>Page Title</TITLE>"
echo " </HEAD>"
echo " <BODY>"
echo " Page body."
echo " </BODY>"
echo "</HTML>"
```

Nuestro primer acercamiento a este problema contiene un shebang, un comentario (siempre es una buena idea) y una secuencia de comandos `echo`, uno por cada línea de salida. Tras guardar el archivo, lo haremos ejecutable y trataremos de ejecutarlo:

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
[me@linuxbox ~]$ sys_info_page
```

Cuando el programa se ejecute, deberíamos ver el texto del documento HTML mostrado en la pantalla, ya que los comandos `echo` del script envían su salida a la salida estándar. Ejecutaremos el programa de nuevo y redirigiremos la salida del programa al archivo `sys_info_page.html`, con lo que podemos ver el resultado con un navegador web:

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
[me@linuxbox ~]$ firefox sys_info_page.html
```

Hasta aquí todo bien.

Cuando escribimos programas, siempre es una buena idea esforzarse en que sean simples y claros. El mantenimiento es más fácil cuando el programa es fácil de leer y comprender, no hace falta mencionar que podemos hacer que el programa sea más fácil de realizar reduciendo la cantidad de escritura. Nuestra versión actual del programa funciona bien, pero podría ser más simple. Podríamos en realidad combinar todos los comandos echo en uno, lo que ciertamente haría más fácil añadir más líneas a la salida del programa. Así que, hagamos ese cambio en nuestro programa:

```
#!/bin/bash
# Program to output a system information page
```

```
echo "<HTML>
<HEAD>
<TITLE>Page Title</TITLE>
</HEAD>
<BODY>
Page body.
</BODY>
</HTML>"
```

Una cadena entre comillas puede incluir nuevas líneas, y por lo tanto contener múltiples líneas de texto. El shell seguirá leyendo el texto hasta que se encuentre las comillas de cierre. Funciona de esta forma también en la línea de comandos:

```
[me@linuxbox ~]$ echo "<HTML>
> <HEAD>
> <TITLE>Page Title</TITLE>
> </HEAD>
> <BODY>
> Page body.
> </BODY>
> </HTML>"
```

El carácter ">" que va delante es el prompt de shell contenido en la variable de shell `PS2`. Aparece siempre que escribamos una sentencia multilínea en el shell. Esta función es un poco oscura ahora mismo, pero más adelante, cuando veamos las sentencias de programación multilínea, se transformará en algo más útil.

Segunda etapa: Añadiendo algunos datos

Ahora que nuestro programa puede generar un documento minimalista, pongamos algunos datos en el informe. Para hacerlo, haremos los siguientes cambios:

```
#!/bin/bash
# Program to output a system information page
echo "<HTML>
<HEAD>
<TITLE>System Information Report</TITLE>
</HEAD>
<BODY>
<H1>System Information Report</H1>
</BODY>
</HTML>"
```

Hemos añadido un título de página y un encabezado al cuerpo del informe.

Variables y constantes

Sin embargo, hay un problema con nuestro script. ¿Ves cómo la cadena "System Information Report" está repetida? Con nuestro diminuto script no es un problema, pero imaginemos que nuestro script fuera realmente largo y que tuviéramos múltiples instancias de esta cadena. Si quisiéramos cambiar el título a otra cosa, habríamos tenido que cambiarlo en múltiples sitios, lo que

podría ser mucho trabajo. ¿Y si pudiéramos arreglar el script para que la cadena sólo apareciera una vez y no múltiples veces? Podría hacer que el mantenimiento en el futuro del script sea más fácil. Aquí tenemos cómo podríamos hacerlo:

```
#!/bin/bash
# Program to output a system information page
title="System Information Report"
echo "<HTML>
<HEAD>
<TITLE>$title</TITLE>
</HEAD>
<BODY>
<H1>$title</H1>
</BODY>
</HTML>"
```

Creando una *variable* llamada `title` y asignándole el valor "System Information Report", podemos sacar ventaja de la expansión de parámetros y colocar la cadena en múltiples lugares.

Pero ¿cómo creamos una variable? Simplemente, la usamos. Cuando el shell encuentra una variable, automáticamente la crea. Esto difiere de muchos lenguajes de programación en los que las variables deben ser *declaradas* o definidas explícitamente antes de usarla. El shell es muy laxo en esto, lo que puede causar algunos problemas. Por ejemplo, considera este escenario ejecutado en la línea de comandos:

```
[me@linuxbox ~]$ foo="yes"
[me@linuxbox ~]$ echo $foo
yes
[me@linuxbox ~]$ echo $fool
[me@linuxbox ~]$
```

Primero asignamos el valor "yes" a la variable `foo`, y luego mostramos su valor con `echo`. A continuación mostramos el valor de la variable mal escrita como "fool" y tenemos un resultado en blanco. Esto es porque el shell ha creado felizmente la variable `fool` cuando la encontró, y le ha dado el valor por defecto de nada, o vacía. Con esto, aprendemos que ¡debemos prestar mucha atención a nuestra escritura! También es importante entender que ha ocurrido realmente en este ejemplo. De nuestro vistazo previo a cómo el shell realiza las expansiones, sabemos que el comando:

```
[me@linuxbox ~]$ echo $foo
```

se somete a expansión de parámetros resultando en:

```
[me@linuxbox ~]$ echo yes
```

Mientras que el comando:

```
[me@linuxbox ~]$ echo $fool
```

se expande a:

```
[me@linuxbox ~]$ echo
```

¡La variable vacía se expande en nada! Esto puede causar estragos en comandos que requieren argumentos. Aquí tenemos un ejemplo:

```
[me@linuxbox ~]$ foo=foo.txt
[me@linuxbox ~]$ foo1=foo1.txt
[me@linuxbox ~]$ cp $foo $fool
cp: missing destination file operand after `foo.txt'
Try `cp --help' for more information.
```

Asignamos valores a dos variables, `foo` y `foo1`. Cuando realizamos un `cp`, pero escribimos mal el nombre del segundo argumento. Tras la expansión, al comando `cp` sólo se le envía un argumento, aunque requiere dos.

Hay algunas reglas sobre los nombres de variables:

1. Los nombres de variables pueden consistir en caracteres alfanuméricos (letras y números) y caracteres guión bajo.
2. El primer carácter de un nombre de variable puede ser tanto una letra o un guión bajo.
3. Los espacios y símbolos de puntuación no están permitidos.

La palabra "variable" implica un valor que cambia, y en muchas aplicaciones, las variables se usan de esta forma. Sin embargo, la variable de nuestra aplicación, `title`, se usa como una *constante*. Una constante es casi como una variable ya que tiene un nombre y contiene un valor. La diferencia es que el valor de una constante no cambia. En una aplicación que realice cálculos geométricos, podríamos definir `PI` como una constante, y asignarle el valor de `3,1415`, en lugar de usar el número literalmente a lo largo de nuestro programa. El shell no hace distinción entre constantes y variables; es más bien a conveniencia del programador. Una convención común es usar letras mayúsculas para designar contantes y minúsculas para variables. Modificaremos nuestro script para cumplir con esta convención:

```
#!/bin/bash
# Program to output a system information page
TITLE="System Information Report For $HOSTNAME"
echo "<HTML>
<HEAD>
<TITLE>$TITLE</TITLE>
</HEAD>
<BODY>
<H1>$TITLE</H1>
</BODY>
</HTML>"
```

Tenemos la oportunidad de mejorar nuestro título añadiendo el valor de la variable de shell `HOSTNAME`. Es el nombre de red de la máquina.

Nota: El shell en realidad proporciona una forma de forzar la inmutabilidad de las constantes, a través del uso del comando incluido `declare` con la opción `-r` (sólo lectura). Si hemos asignado `TITTLE` de esta forma:

```
declare -r TITLE="Page Title"
```

el shell prevendrá cualquier asignación posterior a `TITTLE`. Esta característica es poco usada, pero existe en scripts muy formales.

Asignando valores a variables y constantes

Aquí es donde nuestro conocimiento de las expansiones empieza a dar resultados. Como hemos visto, a las variables se le asignan valores así:

```
variable=valor
```

donde *variable* es el nombre de la variable y *valor* es una cadena. Al contrario de otros lenguajes de programación, el shell no se preocupa por el tipo de datos asignados a una variable; los trata a todos como cadenas. Puedes forzar al shell a restringir la asignación a enteros usando el comando `declare` con la opción `-i`, pero como el configurar las variables como sólo lectura, no se suele hacer.

Fíjate que en una asignación, no debe haber espacios entre el nombre de la variable, el signo igual, y el valor. Entonces ¿en qué puede consistir el valor? Cualquier cosa que pueda expandirse en una cadena:

```
a=z # Assign the string "z" to variable a.
b="a string" # Embedded spaces must be within quotes.
c="a string and $b" # Other expansions such as variables can be
# expanded into the assignment.
d=$(ls -l foo.txt) # Results of a command.
e=$((5 * 7)) # Arithmetic expansion.
f="\t\ta string\n" # Escape sequences such as tabs and newlines.
```

Pueden hacerse múltiples asignaciones de variables en una única línea:

```
a=5 b="a string"
```

Durante la expansión, los nombres de variables pueden ir rodeados de llaves opcionales "{}". Ésto es útil en casos donde una variable puede llegar a ser ambigua para su contexto próximo. Aquí, tratamos de cambiar el nombre de un archivo de `myfile` a `myfile1`, usando una variable:

```
[me@linuxbox ~]$ filename="myfile"
[me@linuxbox ~]$ touch $filename
[me@linuxbox ~]$ mv $filename $filename1
mv: missing destination file operand after `myfile'
Try `mv --help' for more information.
```

Este intento falla porque el shell interpreta el segundo argumento del comando `mv` como una variable nueva (y vacía). El problema puede solucionarse de esta forma:

```
[me@linuxbox ~]$ mv $filename ${filename}1
```

Añadiendo las llaves en los extremos, el shell ya no interpreta el `1` del final como parte del nombre de la variable.

Aprovecharemos esta oportunidad para añadir algunos datos a nuestro informe, digamos la fecha y hora en que fue creado el informe y el nombre de usuario del creador:

```
#!/bin/bash
# Program to output a system information page
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated $CURRENT_TIME, by $USER"
echo "<HTML>
<HEAD>
<TITLE>$TITLE</TITLE>
</HEAD>
<BODY>
<H1>$TITLE</H1>
<P>$TIMESTAMP</P>
</BODY>

</HTML>"
```

Documentos-aquí

Hemos visto dos métodos distintos de mostrar nuestro texto, ambos usando el comando `echo.` Hay una tercera forma llamada *documento-aquí* o *script-aquí*. Un documento-aquí es una forma adicional de redirección I/O en la que incluimos un cuerpo de texto en nuestro script y lo introducimos en la entrada estándar de un comando. Funciona así:

```
comando << token
texto
token
```

donde *comando* es el nombre de un comando que acepta la entrada estándar y *token* es una cadena usada para indicar el final del texto incluido. Modificaremos nuestro script para usar un doucmento-aquí:

```
#!/bin/bash



# Program to output a system information page



TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated $CURRENT_TIME, by $USER"



cat << _EOF_
<HTML>
<HEAD>
<TITLE>$TITLE</TITLE>
</HEAD>
<BODY>
<H1>$TITLE</H1>
```

```
<P>$TIMESTAMP</P>
</BODY>
</HTML>
_EOF_
```

En lugar de usar echo, nuestro script ahora usa cat y un documento-aquí. La cadena _EOF_ (que significa "End Of File" o "Fin del archivo", una convención común) ha sido seleccionada como token, y marca el final del texto incluido. Fíjate que el token debe aparecer sólo y no debe haber espacios delante en la línea.

Entonces ¿cual es la ventaja de usar un documento-aquí? Es principalmente lo mismo que echo, excepto que, por defecto, las comillas sencillas y dobles dentro de los documentos-aquí pierden su significado especial para el shell. Aquí hay un ejemplo en la línea de comandos:

```
[me@linuxbox ~]$ foo="some text"
[me@linuxbox ~]$ cat << _EOF_
> $foo
> "$foo"
> '$foo'
> \$foo
> _EOF_
some text
"some text"
'some text'
$foo
```

Como podemos ver, el shell no presta atención a las comillas. Las trata como caracteres ordinarios. Esto nos permite incluir comillas libremente dentro de un documento-aquí. Podría ser útil para nuestro programa de informes.

Los documentos-aquí pueden usarse con cualquier comando que acepte entrada estándar. En este ejemplo, usamos un documento-aquí para pasar una serie de comandos al programa ftp para descargar un archivo de un servidor FTP remoto:

```
#!/bin/bash



# Script to retrieve a file via FTP



FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-
i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz



ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
```

```
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_
ls -l $REMOTE_FILE
```

Si cambiamos el operador de redirección de "<<" a "<<-" el shell ignorará las caracteres tabulador del principio del documento-aquí. Esto permite que un documento-aquí sea indentado, lo que puede mejorar su legibilidad:

```
#!/bin/bash



# Script to retrieve a file via FTP



FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-
i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz



ftp -n <<- _EOF_
  open $FTP_SERVER
  user anonymous me@linuxbox
  cd $FTP_PATH
  hash
  get $REMOTE_FILE
  bye
  _EOF_




ls -l $REMOTE_FILE
```

Resumiendo

En este capítulo, hemos comenzado un proyecto que nos llevará a través del proceso de construcción de un script exitoso. Hemos presentado el concepto de variables y constantes y como pueden utilizarse. Son la primera de las muchas aplicaciones que encontraremos para expansión de parámetros. También hemos visto como producir salida de nuestro script, y varios métodos para incluir bloques de texto.

Para saber más

- Para más información sobre HTML, mira los siguientes artículos y tutoriales:

http://en.wikipedia.org/wiki/Html

http://en.wikibooks.org/wiki/HTML_Programming
http://html.net/tutorials/html/

- La man page de `bash` incluye una sección titulada "Documentos-aquí", que tiene una descripción completa de esta característica.

Diseño de arriba a abajo

A medida que los programas se hacen más grandes y complejos, se hacen más difíciles de diseñar, codificar y mantener. Al iguar que con cualquier gran proyecto, es a menudo una buena idea dividir las tareas grandes y complejas en una serie de tareas simples y pequeñas. Imaginemos que estamos intentando describir un tarea común de todos los días, ir al mercado a comprar comida, para una persona de Marte. Describiríamos el proceso completo como la siguiente serie de pasos:

1. Subir al coche
2. Conducir hasta el mercado
3. Aparcar el coche
4. Entrar en el mercado
5. Comprar comida
6. Volver al coche
7. Conducir a casa
8. Aparcar el coche
9. Entrar en casa

Sin embargo una persona de Marte seguro que necesita más detalles. Podríamos dividir aún más las subtarea "Aparcar el coche" en una serie de pasos:

1. Encontrar un sitio para aparcar
2. Meter el coche dentro de ese sitio
3. Apagar el motor
4. Poner el freno de mano
5. Salir del coche
6. Cerrar el coche

La subtarea "Apagar el motor" podría dividirse aún más en pasos incluyendo "Apagar el contacto", "Sacar la llave", y así sucesivamente, hasta que cada paso del proceso completo de ir al mercado hubiera sido completamente definido.

El proceso de identificar los pasos de alto nivel y desarrollar incrementalmente vistas detalladas de dichos pasos se llama *diseño de arriba a abajo*. Esta técnica nos permite romper tareas grandes y complejas en muchas tareas pequeñas y simples. El diseño de arriba a abajo es un método común de diseñar programas y uno de los que se ajustan muy bien a la programación en shell en particular.

En este capítulo, usaremos el diseño de arriba a abajo para seguir desarrollando nuestro script generador de informes.

Funciones de shell

Nuestro script actualmente realiza los siguientes pasos para general el documento HTML:

1. Abre la página
2. Abre el encabezado de página.
3. Establece el título de página.
4. Cierra el encabezado de página.
5. Abre el cuerpo de la página.

6. Muestra el encabezado de la página.
7. Muestra la hora.
8. Cierra el cuerpo de la página.
9. Cierra la página.

Para nuestra próxima fase de desarrollo, añadiremos algunas tareas entre los pasos 7 y 8. Estos incluirán:

- Hora de encendido y carga del sistema. Es la cantidad de tiempo desde el último encendido o reinicio y el número medio de tareas corriendo actualmente en el procesador en varios intervalos de tiempo.
- Espacio en disco. El uso total de espacio en los dispositivos del sistema de almacenamiento.
- Espacio Home. La cantidad de espacio de almacenamiento usado por cada usuario.

Si tuviéramos un comando para cada una de dichas tareas, podríamos añadirlos a nuestro script simplemente a través de sustitución de comandos:

```
#!/bin/bash



# Program to output a system information page



TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated $CURRENT_TIME, by $USER"



cat << _EOF_
<HTML>
<HEAD>
<TITLE>$TITLE</TITLE>
</HEAD>
<BODY>
<H1>$TITLE</H1>
<P>$TIMESTAMP</P>
$(report_uptime)
$(report_disk_space)
$(report_home_space)
</BODY>
</HTML>
_EOF_
```

Podríamos crear estos comandos adicionales de dos formas. Podríamos escribir tres scripts separados y colocarlos en un directorio listado en nuestro PATH, o podríamos incluir los scripts dentro de nuestro programa como *funciones de shell*. Como hemos mencionado antes, las funciones de shell son "mini-scripts" que se colocan dentro de otros scripts y pueden actuar como programas autónomos. Las funciones de shell tienen dos formas sintácticas:

```
function name {
commands
```

```
return
}
and
name () {
commands
return
}
```

donde *name* es el nombre de la función y *commands* es una serie de comandos contenidos dentro de la función. Ambas formas son equivalentes y podrían usarse indistintamente. A continuación vemos un script que demuestra el uso de una función de shell:

```
1 #!/bin/bash
2
3 # Shell function demo
4
5 function funct {
6 echo "Step 2"
7 return
8 }
9
10 # Main program starts here
11
12 echo "Step 1"
13 funct
14 echo "Step 3"
```

A medida que el shell lee el script, va pasando por alto de la línea 1 a la 11, ya que estas líneas son comentarios y la definición de la función. La ejecución comienza en la línea 12 con un comando `echo`. La línea 13 llama a la función de shell `funct` y el shell ejecuta la función tal como haría con cualquier otro comando. El control del programa se mueve entonces a la línea 6, y se ejecuta el segundo comando `echo`. La línea 7 se ejecuta a continuación. Su comando `return` finaliza la función y devuelve el control al programa en la línea que sigue a la función call (línea 14), y el comando final `echo` se ejecuta. Fíjate que, para que las llamadas a funciones sean reconocidas como funciones de shell y no interpretadas como nombres de programas externos, las definiciones de funciones de shell deben aparecer antes de que sean llamadas.

Añadiremos unas definiciones de funciones mínimas a nuestro script:

```
#!/bin/bash



# Program to output a system information page



TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated $CURRENT_TIME, by $USER"
```

```
report_uptime () {
return
}
report_disk_space () {
return
}
report_home_space () {
return
}
cat << _EOF_
<HTML>
<HEAD>
<TITLE>$TITLE</TITLE>
</HEAD>
<BODY>
<H1>$TITLE</H1>
<P>$TIMESTAMP</P>
$(report_uptime)
$(report_disk_space)
$(report_home_space)
</BODY>
</HTML>
_EOF_
```

Los nombres de funciones de shell siguen las mismas reglas que las variables. Una función debe contener al menos un comando. El comando `return` (que es opcional) satisface este requerimiento.

Variables locales

En los scripts que hemos escrito hasta ahora, todas las variables (incluyendo las constantes) han sido *variables globales*. Las variables globales siguen existiendo a lo largo del programa. Esto es bueno para muchas cosas, pero, a veces, puede complicar el uso de las funciones de shell. Dentro de las funciones de shell, a menudo es preferible tener *variables locales*. Las variables locales solo son accesibles dentro de la función de shell en la que han sido definidas y dejan de existir una vez que la función de shell termina.

Tener variables locales permite al programador usar variables con nombres que pueden existir anteriormente, tanto en el script globalmente o en otras funciones de shell, si tener que preocuparnos por potenciales confilctos de nombres.

Aquí hay un ejemplo de script que demuestra como se definen y usan las variables locales:

```
#!/bin/bash



# local-vars: script to demonstrate local variables
foo=0 # global variable foo
funct_1 () {
local foo # variable foo local to funct_1
```

```
foo=1
echo "funct_1: foo = $foo"
}
funct_2 () {
local foo # variable foo local to funct_2
foo=2
echo "funct_2: foo = $foo"
}
echo "global: foo = $foo"
funct_1
echo "global: foo = $foo"
funct_2
echo "global: foo = $foo"
```

Como podemos ver, las variables locales se definen precediendo al nombre de la variable con la palabra `local`. Ésto crea un variable que es local para la función de shell en la que se define. Una vez fuera de la función de shell, la variable deja de existir. Cuando ejecutamos este script, vemos los resultados:

```
[me@linuxbox ~]$ local-vars
global: foo = 0
funct_1: foo = 1
global: foo = 0
funct_2: foo = 2
global: foo = 0
```

Vemos que la asignación de valores a la variable local `foo` dentro de ambas funciones de shell no tiene efecto en el valor de `foo` definido fuera de las funciones.

Esta característica permite que las funciones de shell sean escritas de forma que se mantengan independientes una de otra y del script en el que aparecen. Esto es muy valioso, ya que ayuda a prevenir que una parte del programa interfiera en otra. También permite que las funciones de shell sean escritas de forma que sean portables. O sea, pueden cortarse y pegarse de un script a otro, como sea necesario.

Mantener los scripts ejecutándose

Mientras que desarrollamos nuestro programa, es útil mantener el programa en estado ejecutable. Haciendo esto, y probándolo frecuentemente, podemos detectar errores pronto en el proceso de desarrollo. Esto hará los problemas de depurado más fáciles. Por ejemplo, si ejecutamos el programa, hacemos un pequeño cambio, y ejecutamos el programa de nuevo, es muy probable que el cambio más reciente sea la fuente del problema. Añadiendo las funciones vacías, llamadas stubs en el habla de los programadores, podemos verificar el flujo lógico de nuestro programa en una fase temprana. Cuando construimos un stub, es una buena idea incluir algo que proporcione retroalimentación al programador, que muestre que el flujo lógico se está realizando. Si miramos la salida de nuestro script ahora:

```
[me@linuxbox ~]$ sys_info_page
<HTML>
<HEAD>
<TITLE>System Information Report For twin2</TITLE>
</HEAD>
```

```
<BODY>
<H1>System Information Report For linuxbox</H1>
<P>Generated 03/19/2009 04:02:10 PM EDT, by me</P>









</BODY>
</HTML>
```

vemos que hay algunas líneas en blanco en nuestra salida tras la marca de tiempo, pero no podemos estár seguros de cual es la causa. Si cambiamos las funciones para incluir algo de retroalimentación:

```
report_uptime () {
echo "Function report_uptime executed."
return
}
report_disk_space () {
echo "Function report_disk_space executed."
return
}
report_home_space () {
echo "Function report_home_space executed."
return
}
```

y ejecutamos el script de nuevo:

```
[me@linuxbox ~]$ sys_info_page
<HTML>
<HEAD>
<TITLE>System Information Report For linuxbox</TITLE>
</HEAD>
<BODY>
<H1>System Information Report For linuxbox</H1>
<P>Generated 03/20/2009 05:17:26 AM EDT, by me</P>
Function report_uptime executed.
Function report_disk_space executed.
Function report_home_space executed.
</BODY>
</HTML>
```

ahora vemos, de hecho, que nuestras tres funciones se están ejecutando.

Con nuestra estrucutra de funciones en su sitio y funcionando, es hora desarrollar el código de nuestras funciónes. Primero, la `función report_uptime`:

```
report_uptime () {
```

```
cat <<- _EOF_
<H2>System Uptime</H2>
<PRE>$(uptime)</PRE>
_EOF_
return
}
```

Es bastante sencilla. Usamos un documento-aquí para mostrar un encabezado de sección y la salida del comando `uptime`, rodeándolos de etiquetas `<PRE>` para evitar el formateo del comando. La función `report_disk_space` es similar:

```
report_disk_space () {
cat <<- _EOF_
<H2>Disk Space Utilization</H2>
<PRE>$(df -h)</PRE>
_EOF_
return
}
```

Esta función usa el comando `df -h` para determinar la cantidad de espacio en disco. Finalmente, construiremos la función `report_home_space`:

```
report_home_space () {
cat <<- _EOF_
<H2>Home Space Utilization</H2>
<PRE>$(du -sh /home/*)</PRE>
_EOF_
return
}
```

Usamos el comando `du` con las opciones `-sh` para realizar esta tarea. Esto, sin embargo, no es una solución completa al problema. Aunque funcionará en algunos sitemas (Ubuntu, por ejemplo), no funcionará en otros. La razón es que muchos sitemas establecen los permisos de los directorios home para prevenir que sean legibles por todo el mundo, lo que es una medida de seguridad razonable. En estos sistemas, la función `report_home_space`, tal como está escrita, sólo funcionará si nuestro script se ejecuta con privilegios de superusuario. Un solución mejor sería hacer que el script ajuste su comportamiento según los privilegios del usuario. Lo veremos en el próximo capítulo.

Funciones shell en su archivo .bashrc

Las funciones de shell hacen excelentes reemplazos para los alias, y son en realidad el método preferido de crear pequeños comandos para uso personal. Los alias están muy limitados por el tipo de comandos y características de shell que soportan, mientras que las funciones de shell permiten cualquier cosa que pueda ser incluida en un script. Por ejemplo, si nos gusta la función shell `report_disk_space` que hemos desarrollado en nuestro script, podríamos crear una función similar llamada `ds` para nuestro archivo `.bashrc`:

```
ds () {
echo “Disk Space Utilization For $HOSTNAME”
df -h
```

```
}
```

Resumiendo

En este capítulo, hemos presentado un método común de diseño de programas llamado diseño de arriba a abajo, y hemos visto cómo las funciones de shell se usan para construir el refinamiento paso a paso que requiere. También hemos visto cómo las variables locales pueden usarse para hacer a las funciones de shell independientes unas de otras y del programa en las que se encuentran. Esto hace posible que las funciones de shell sean escritas de una forma portable y que sean *reutilizables* permitiendo que se coloquen en múltiples programas; un gran ahorro de tiempo.

Para saber más

- La Wikipedia tiene muchos artículos sobre filosofía de desarrollo de software. Aquí tienes un par de ellos muy buenos:

http://en.wikipedia.org/wiki/Top-down_design
http://en.wikipedia.org/wiki/Subroutines

Control de flujo: Ramificando con if

En el último capítulo, nos hemos encontrado con un problema. ¿Cómo podemos hacer que nuestro script generador de informes se adapte a los privilegios del usuario que está ejecutando el script? La solución a este problema requerirá que encontremos una forma de "cambiar direcciones" dentro de nuestro script, basándose en el resultado de un test. En términos de programación, necesitamos un programa para *ramificar*.

Consideremos un ejemplo simple de lógica expresada en *pseudocódigo*, un lenguaje de ordenador simulado pensado para el consumo humano:

```
X = 5
Si X = 5, entonces:
Dí “X es igual a 5.”
Si nó::
Dí “X no es igual a 5.”
```

Este es un ejemplo de ramificación. Basada en la condición "¿Es X = 5?" haz una cosa "Dí X es igual a 5", si no haz otra cosa, "Dí X no es igual a 5."

if

Usando el shell, podemos codificar la lógica anterior tal como sigue:

```
x=5
if [ $x -eq 5 ]; then
echo "x equals 5."
else
echo "x does not equal 5."
fi
```

o podemos introducirlo directamente en la línea de comandos (lígeramente simplificada):

```
[me@linuxbox ~]$ x=5
[me@linuxbox ~]$ if [ $x -eq 5 ]; then echo "equals 5"; else echo
"does not equal 5"; fi
equals 5
[me@linuxbox ~]$ x=0
[me@linuxbox ~]$ if [ $x -eq 5 ]; then echo "equals 5"; else echo
"does not equal 5"; fi
does not equal 5
```

En este ejemplo, ejecutamos el comando dos veces. La primera, con el valor de `x` establecido en `5`, que da como resultado la cadena "equals 5" como salida, y la segunda vez con el valor de `x` establecido en `0`, que da como resultado la cadena "does not equal 5" como salida.

La sentencia `if` tiene la siguiente sintaxis:

```
if comandos; then
comandos
[elif comandos; then
comandos...]
[else
comandos]
fi
```

donde *comandos* es una lista de comandos. Ésto es un poco confuso a primera vista. Pero antes de que podamos aclararlo, tenemos que ver cómo evalua el shell el éxito o error de un comando.

Estado de la salida

Los comandos (incluyendo los scripts y las funciones de shell que escribamos) envían un valor al sistema cuando terminan, llamado *estado de la salida*. Este valor, que es un entero dentro del rango de 0 a 255, indica el éxito o fracaso de la ejecución del comando. Por consenso, un valor de cero indica éxito y cualquier otro valor indica fracaso. El shell proporciona un parámetro que puede usarse para examinar el estado de la salida. Aquí lo vemos en acción:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0 [
me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

En este ejemplo, ejecutamos el comando `ls` dos veces. La primera vez, el comando se ejecuta con éxito. Si mostramos el valor del parámetro `$?`, vemos que es un cero. Ejecutamos el comando `ls` una segunda vez, produciendo un error, y examinamos el parámetro `$?` otra vez. Esta vez contiene un 2, indicando que el comando ha encontrado un error. Algunos comandos usan estados de salida diferentes para proporcionar diagnósticos de error, meientras que muchos comandos simplemente salen con el valor uno cuando se usan. Sin embargo, un cero siempre indica éxito.

El shell proporciona dos comandos extremadamente simples que no hacen nada excepto terminar con un cero o un uno en la salida de estado. El comando `true` siempre se ejecuta con éxito y el comando `false` siempre se ejecuta sin éxito:

```
[me@linuxbox ~]$ true
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ false
[me@linuxbox ~]$ echo $?
1
```

Podemos usar estos comandos para ver como funcionan las sentencias `if`. Lo que la sentencia `if` hace en realidad es evaluar el éxito o el fracaso de los comandos:

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

El comando echo "It's true." se ejecuta cuando el comando que sigue a `if` se ejecuta con éxito, y no se ejecuta cuando el comando que sigue a `if` no se ejecuta con éxito. Si es una lista de comandos lo que sigue a `if`, el que se evalua es el último comando de la lista:

```
[me@linuxbox ~]$ if false; true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if true; false; then echo "It's true."; fi

[me@linuxbox ~]$
```

test

De lejos, el comando usado más frecuentemente con `if` es `test`. El comando `test` realiza una variedad de comprobaciones y comparaciones. Tiene dos formas equivalentes:

```
test expresión
```

y la más popular:

```
[ expresión ]
```

donde *expresión* es una expresión que se evalua tanto si es falsa como si es verdadera. El comando test devuelve un estado de salida de cero cuando la expresión es verdadera y de uno cuando la expresión es falsa.

Expresiones para archivo

Las siguientes expresiones se usan para evaluar el estado de los archivos.

Tabla 27-1: Expresiones test para archivos

Expresión	Es verdadero si:

archivo1 -ef *archivo2*	*archivo1* y *archivo2* tienen los mismos números de inodo (los dos nombres de archivo se refieren al mismo archivo por enlace duro).
archivo1 -nt *archivo2*	*archivo1* es más nuevo que *archivo2*.
archivo1 -ot *archivo2*	*archivo1* es más antiguo que *archivo2*.
-b *archivo*	*archivo* existe y es un archivo con bloqueo especial (dispositivo).
-c *archivo*	*archivo* existe y es un archivo de caracter especial (dispositivo).
-d *archivo*	*archivo* existe y es un directorio.
-e *archivo*	*archivo* existe.
-f *archivo*	*archivo* existe y es un archivo normal.
-g *archivo*	*archivo* exite y tiene establecida una ID de grupo.
-G *archivo*	*archivo* exite y su propietario es el ID de grupo efectivo.
-k *archivo*	*archivo* exite y tiene establecido su "sticky bit"
-L *archivo*	*archivo* existe y es un enlace simbólico.
-O *archivo*	*archivo* existe y su propietario es el ID de usuario efectivo.
-p *archivo*	*archivo* existe y es un entubado con nombre.
-r *archivo*	*archivo* existe y es legible (tiene permisos de lectura para el usuario efectivo).
-s *archivo*	*archivo* existe y tiene una longitud mayor que cero.
-S *archivo*	*archivo* existe y es una conexión de red
-t *fd*	*fd* es un descritor de archivo dirigido de/hacia el terminal. Puede usarse para determinar que entrada/salida/error estándar está siendo redirigido.
-u *archivo*	*archivo* existe y es setuid
-w *archivo*	*archivo* existe es es editable (tiene permisos de escritura para el usuario efectivo).
-x *archivo*	*archivo* existe y es ejecutable (tiene permisos de ejecución/búsqueda para el usuario efectivo).

Aquí tenemos un script que demuestra algunas de las expresiones para archivo:

```
#!/bin/bash
# test-file: Evaluate the status of a file
FILE=~/.bashrc
```

```
if [ -e "$FILE" ]; then
if [ -f "$FILE" ]; then
echo "$FILE is a regular file."
fi
if [ -d "$FILE" ]; then
echo "$FILE is a directory."
fi
if [ -r "$FILE" ]; then
echo "$FILE is readable."
fi
if [ -w "$FILE" ]; then
echo "$FILE is writable."
fi
if [ -x "$FILE" ]; then
echo "$FILE is executable/searchable."
fi
else
echo "$FILE does not exist"
exit 1
fi
exit
```

El script evalua el archivo asignado a la constante `FILE` y muestra su resultado una vez que se realiza la evaluación. Hay dos cosas interesantes a tener en cuenta sobre este script. Primero, fíjate como el parámetro `$FILE` está entrecomillado junto con las expresiones. Esto no se requiere, pero es una defensa contra un parámetro vacío. Si la expansión de parámetros de `$FILE` fuera a dar como resultado un valor vacío, podría causar un error (los operadores serían interpretados como cadenas no nulas en lugar de como operadores). Usando las comillas alrededor de los parámetros nos asegurarmos que el operador siempre está seguido por una cadena, incluso si la cadena está vacía. Segundo, fíjate en la presencia del comando `exit` cerca del final del script. El comando `exit` acepta un úncio, y opcional, argumento que se convierte en el estado de salida del script. Cuando no se le pasa ningún argumento, el estado de salida por defecto es la salida el último comando ejecutado. Usar `exit` de esta forma permite que el script indique fallo si `$FILE` se expande en el nombre de un archivo inexistente. El comando `exit` que aparece en la última línea del script es una formalidad aquí. Cuando un script "alcanza el final" (llega al final del archivo), termina con un estado de salida del último comando ejecutado por defecto, de todas formas.

Similarmente, las funciones de shell pueden devolver un estado de salida incluyendo un argumento entero al comando `return`. Si fueramos a convertir el script siguiente en una función para incluirlo en un progrma mayor, podríamos reemplazar los comandos `exit` con sentencias `return` y obtener el comportamiento deseado:

```
test_file () {
# test-file: Evaluate the status of a file
FILE=~/.bashrc
if [ -e "$FILE" ]; then
if [ -f "$FILE" ]; then
echo "$FILE is a regular file."
fi
if [ -d "$FILE" ]; then
echo "$FILE is a directory."
fi
```

```
if [ -r "$FILE" ]; then
echo "$FILE is readable."
fi
if [ -w "$FILE" ]; then
echo "$FILE is writable."
fi
if [ -x "$FILE" ]; then
echo "$FILE is executable/searchable."
fi
else
echo "$FILE does not exist"
return 1
fi

}
```

Expresiones para cadenas

Las siguientes expresiones se usan para evaluar cadenas:

Tabla 27-2: Expresiones para test de cadenas

Expresión	**Es verdadero si:**
cadena	*cadena* no es nula.
-n *cadena*	La longitud de *cadena* es mayor que cero.
-z *cadena*	La longitud de *cadena* es cero.
cadena1 = *cadena2* *cadena1* == *cadena2*	*cadena1* y *cadena2* son iguales. Pueden usarse signos igual simples o dobles, pero es preferible usar dobles signos igual.
cadena1 != *cadena2*	*cadena1* y *cadena2* no son iguales.
cadena1 > *cadena2*	*cadena1* se ordena detrás de *cadena2*.
cadena1 < *cadena2*	*cadena1* se ordena antes de *cadena2*.

Advertencia: Los operadores de expresión > y < deben ser entrecomillados (o escapados con una barra invertida) cuando se usan con `test`. Si no se hace esto, serán interpretados por el shell como operadores de redirección, con resultados potencialmente destructivos. Fíjate también que mientras que la documentación `bash` establece que el ordenado se hace según el locale actual, no lo hace así. Se usa el orden ASCII (POSIX) en versiones de `bash` hasta la 4.0 incluida.

Aquí tenemos un script que incorpora expresiones para cadenas:

```
#!/bin/bash
# test-string: evaluate the value of a string
ANSWER=maybe
if [ -z "$ANSWER" ]; then
echo "There is no answer." >&2
exit 1
fi
```

```
if [ "$ANSWER" = "yes" ]; then
echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
echo "The answer is MAYBE."
else
echo "The answer is UNKNOWN."
fi
```

En este script, evaluamos la constante `ANSWER`. Primero determinamos si la cadena está vacía. Si es así, terminamos el script y establecemos el estado de salida a uno. Fíjate la redirección que se aplica al comando echo. Redirige el mensaje de error "No hay respuesta." al error estándar, que es lo "apropiado" que hay que hacer con los mensajes de error. Si la cadena no está vacía, evaluamos el valor de la cadena para ver si es igual a "sí", "no" o "quizás". Hacemos esto usando `elif`, que es la abreviatura para "else if" (y si no). Usando `elif`, podemos construir un test lógico más complejo.

Expresiones con enteros

Las siguientes expresiones se usan con enteros:

Tabla 27-3: Expresiones de test con enteros

Expresión	**Es verdadero si:**
entero1 -eq *entero2*	*entero1* es igual a *entero2*.
entero1 -ne *entero2*	*entero1* no es igual a *entero2*.
entero1 -le *entero2*	*entero1* es menor o igual a *entero2*.
entero1 -lt *entero2*	*entero1* es menor que *entero2*.
entero1 -ge *entero2*	*entero1* es mayor o igual a *entero2*.
entero1 -gt *entero2*	*entero1* es mayor que *entero2*.

Aquí hay un script que lo demuestra:

```
#!/bin/bash
# test-integer: evaluate the value of an integer.
INT=-5
if [ -z "$INT" ]; then
echo "INT is empty." >&2
exit 1
fi
if [ $INT -eq 0 ]; then
echo "INT is zero."
else
if [ $INT -lt 0 ]; then
echo "INT is negative."
else
echo "INT is positive."
```

```
fi
if [ $((INT % 2)) -eq 0 ]; then
echo "INT is even."
else
echo "INT is odd."
fi
fi
```

La parte interesante del script es como determina si es un entero es par o impar. Realizando una operación de módulo 2 con el número, que divide el número en dos y devuelve el resto, puede decirnos si es par o impar.

Una versión más moderna de test

Versiones recientes de `bash` incluyen un comando compuesto que actúa como reemplazo mejorado de `test`. Usa la siguiente sintaxis:

`[[ ` *`expresión`* ` ]]`

donde, como en `test`, *expresión* es una expresión que evalúa si un resultado es verdadero o falso. El comando [[ ]] es muy parecido a `test` (soporta todas sus expresiones), pero añade una nueva expresión de cadena importante:

`cadena1 =~ regex`

que devuelve si *cadena1* está marcada con la expresión regular extendida *regex*. Esto abre un montón de posibilidades para realizar tareas tales como validación de datos. En nuestro ejemplo anterior de las expresiones con enteros, el script fallaría si la constante `INT` contiene cualquier cosa excepto un entero. El script necesita una forma de verificar si la constante contiene un entero. Usando [[ ]] con el operador de expresiónes de cadena =~, podría mejorar el script de esta forma:

```
#!/bin/bash
# test-integer2: evaluate the value of an integer.
INT=-5
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
if [ $INT -eq 0 ]; then
echo "INT is zero."
else
if [ $INT -lt 0 ]; then
echo "INT is negative."
else
echo "INT is positive."
fi
if [ $((INT % 2)) -eq 0 ]; then
echo "INT is even."
else
echo "INT is odd."
fi
fi
else
echo "INT is not an integer." >&2
exit 1
```

```
fi
```

Aplicando la expresión regular, podemos limitar el valor de `INT` a sólo cadenas que comiencen con un signo menos opcional, seguido de uno o más números. Esta expresión también elimina la posibilidad de valores vacíos.

Otra característica añadida de [[ ]] es que el operador == soporta coincidencias de patrones de la misma forma que lo hace la expansión. Por ejemplo:

```
[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'
```

Esto hace a [[ ]] útil para evaluar archivos y rutas.

(( )) - Diseñado para enteros

Además del comando compuesto `[[ ]]`, `bash` también proporciona el comando compuesto `(( ))`, que es útil para operar con enteros. Soporta una serie completa de evaluaciones aritméticas, un asunto que veremos a fondo en el Capítulo 34.

`(( ))` se usa para realizar *pruebas de veracidad aritmética*. Una prueba de veracidad aritmética es verdadera si el resultado de la evaluación aritmética no es cero.

```
[me@linuxbox ~]$ if ((1)); then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ if ((0)); then echo "It is true."; fi
[me@linuxbox ~]$
```

Usando `(( ))`, podemos simplificar ligéramente el script `test-integer2` así:

```
#!/bin/bash
# test-integer2a: evaluate the value of an integer.
INT=-5
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
if ((INT == 0)); then
echo "INT is zero."
else
if ((INT < 0)); then
echo "INT is negative."
else
echo "INT is positive."
fi
if (( ((INT % 2)) == 0)); then
echo "INT is even."
else
echo "INT is odd."
fi
fi
```

```
else
echo "INT is not an integer." >&2
exit 1
fi
```

Fíjate que usamos los signos menor-que y mayor-que y que `==` se usa para comprobar equivalencias. Esta es una sintaxis con un aspecto más natural para trabajar con enteros. Fíjate también, que ya que el comando compuesto `(( ))` es parte de la sintaxis del shell en lugar de un comando ordinario, que sólo trabaja con enteros, puede reconocer variables por el nombre y no requiere expansión para hacerlo. Veremos (( )) y la expansión aritmética relacionada más adelante en el Capítulo 34.

Combinando expresiones

También es posible combinar expresiones para crear evaluaciones más complejas. Las expresiones se combinan usando operadores lógicos. Los vimos en el Capítulo 17, cuando aprendimos el comando `find`. Hay tres operaciones lógicas para `test` y `[[ ]]`. Son AND, OR y NOT. test y `[[ ]]` usan operadores diferentes para representar estas operaciones:

Tabla 27-4: Operadores lógicos

Operación	test	[[ ]] y (( ))
AND	-a	&&
OR	- o	| |
NOT	!	!

Aquí tenemos un ejemplo de una operación AND. El siguiente script determina si un entero está dentro de un rango de valores:

```
#!/bin/bash
# test-integer3: determine if an integer is within a
# specified range of values.
MIN_VAL=1
MAX_VAL=100
INT=50
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
if [[ INT -ge MIN_VAL && INT -le MAX_VAL ]]; then
echo "$INT is within $MIN_VAL to $MAX_VAL."
else
echo "$INT is out of range."
fi
else
echo "INT is not an integer." >&2
exit 1
fi
```

En este script, determinamos si el valor del entero `INT` se encuentra entre los valores `MIN_VAL` y `MAX_VAL`. Esto se realiza con un simple uso de `[[ ]]`, que incluya dos expresiones separadas por

el operador &&. Podríamos haber realizado este código usando `test`:

```
if [ $INT -ge $MIN_VAL -a $INT -le $MAX_VAL ]; then
echo "$INT is within $MIN_VAL to $MAX_VAL."
else
echo "$INT is out of range."
fi
```

El operador de negación ! invierte la salida de una expresión. Devuelve verdadero si una expresión es falsa, y devuelve falso si una expresión es verdadera. En el siguiente script, modificamos la lógica de nuestra evaluación para encontrar valores de `INT` que están fuera del rango especificado:

```
#!/bin/bash
# test-integer4: determine if an integer is outside a
# specified range of values.
MIN_VAL=1
MAX_VAL=100
INT=50
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
if [[ ! (INT -ge MIN_VAL && INT -le MAX_VAL) ]]; then
echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
echo "$INT is in range."
fi
else
echo "INT is not an integer." >&2
exit 1
fi
```

También incluimos paréntesis alrededor de la expresión, para agruparlo. Si no se incluyeran, la negación sólo se aplicaría a la primera expresión y no a la combinación de las dos. Para escribir el código con `test` lo haríamos de esta forma:

```
if [ ! \( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \) ]; then
echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
echo "$INT is in range."
fi
```

Como todas las expresiones y operaciones usadas con `test` se tratan como argumentos de comandos por el shell (al contrario de `[[ ]]` y `(( ))`), que son caracteres que tienen un significado especial para bash, como `<`, `>`, `(`, y `(`, deben ser entrecomillados o escapados.

Viendo que `test` y `[[ ]]` hacen más o menos lo mismo, ¿cual es mejor? `test` es tradicional (y parte de POSIX), mientras que `[[ ]]` es especifico para `bash`. Es importante saber como usar `test`, ya que su uso está muy expandido, pero `[[ ]]` es claramente más útil y fácil para escribir código.

La portabilidad es el duende de las mentes pequeñas

Si hablas con gente del Unix real, descubrirás rápidamente que a muchos de ellos no les gusta mucho Linux. Lo consideran impuro y sucio. Un principio de los seguidores de Unix es que todo

debe ser "portable". Esto significa que cualquier script que escribas debería poder funcionar, sin cambios, en cualquier sistema como-Unix.

La gente de Unix tiene una buena razón para creer esto. Han visto lo que las extensiones propietarias de los comandos y del shell hicieron al mundo Unix antes de POSIX, están naturalmente escarmentados del efecto de Linux en su amado SO.

Pero la portabilidad tiene un serio inconveniente. Impide el progreso. Requiere que las cosas se hagan siempre usando técnicas de "mínimo común denominador". En el caso de la programación en shell, significa hacerlo todo compatible con sh, el Bourne shell original.

Este inconveniente es la excusa que usan los distribuidores propietarios para justificar sus extensiones propietarias, sólo que las llaman "innovaciones". Pero realmente sólo están capando dispositivos para sus clientes.

Las herramientas GNU, como `bash`, no tienen estas restricciones. Consiguen la portabilidad soportando estándares y estando disponibles universalmente. Puedes instalar `bash` y las otras herramientas GNU en casi todo tipo de sistema, incluso en Windows, sin coste. Así que siéntete libre para usar todas las características de `bash`. Es *realmente* portable.

Operadores de control: otra forma de ramificar

`bash` ofrece dos operadores de control que pueden realizar ramificaciones. Los operadores && (AND) y `||` (OR) funcionan como los operadores lógicos del comando compuesto `[[ ]]`. Esta es la sintaxis:

`comando1 && comando2`

y

`comando1 || comando2`

Es importante entender este comportamiento. Con el operador &&, se ejecuta *comando1,* y *comando2* se ejecuta si, y sólo si, *comando1* es exitoso. Con el operador `||`, se ejecuta *comando1,* y *comando2* se ejecuta si, y sólo si, *comando2* no es exitoso.

En términos prácticos, esto significa que podemos hacer algo así:

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

Esto creará un directorio llamado `temp`, y si tiene éxito, el directorio de trabajo actual se cambiará a `temp`. El segundo comando sólo se intenta ejecutar si el comando `mkdir` tiene éxito. Igualmente, un comando como éste:

```
[me@linuxbox ~]$ [ -d temp ] || mkdir temp
```

comprobaremos si existe el directorio `temp`, y sólo si falla el test, se creará el directorio. Este tipo de construcción es muy útil para detectar errores en scripts, y asunto que se tratará más tarde en capítulos posteriores. Por ejemplo, podríamos hacer esto en un script:

```
[ -d temp ] || exit 1
```

Si el script requiere el directorio `temp`, y no existe, entonces el script terminará con un estado de salida de uno.

Resumiendo

Hemos comenzado este capítulo con una pregunta. ¿Cómo podríamos hacer que nuestro script `sys_info_page` detecte si el usuario tiene permisos para leer todos los directorios home? Con nuestro conocimiento de `if`, podemos resolver el problema añadiendo este código a la función `report_home_space`:

```
report_home_space () {
if [[ $(id -u) -eq 0 ]]; then
cat <<- _EOF_
<H2>Home Space Utilization (All Users)</H2>
<PRE>$(du -sh /home/*)</PRE>
_EOF_
else
cat <<- _EOF_
<H2>Home Space Utilization ($USER)</H2>
<PRE>$(du -sh $HOME)</PRE>
_EOF_
fi
return
}
```

Evaluamos la salida del comando `id`. Con la opción `-u`, `id` muestra el ID numérico del usuario efectivo. El superusuario siempre es cero y cualquier otro usuario es un número mayor de cero. Sabiendo esto, podemos construir dos documentos-aquí diferentes, uno con las ventajas de los privilegios del superusuario, y el otro, restringido a directorio home del usuario.

Vamos a tomarnos un descanso del programa `sys_info_page`, pero no te preocupes. Volverá. Mientras, veremos algunos temas que necesitaremos cuando retomemos nuestro trabajo.

Para saber más

Hay varias secciones de la man page de `bash` que ofrecen más detalles de los temas que hemos visto en este capítulo:

- Listas (trata los operadores de control `||` y &&)
- Comandos compuestos (trata `[[ ]]`, `(( ))` e `if`)
- EXPRESIONES CONDICIONALES
- COMANDOS INCLUIDOS EN EL SHELL (incluye `test`)

Además, la Wikipedia tiene un buen artículo sobre el concepto de pseudocódigo:

http://en.wikipedia.org/wiki/Pseudocode

Leyendo la entrada del teclado

Los scripts que hemos escrito hasta ahora carecen de una característica común a la mayoría de los

programas informáticos - *interactividad*. O sea, la capacidad de los programas para interactuar con el usuario. Aunque la mayoría de los programas no necesitan ser interactivos, algunos programas se benefician de poder aceptar entrada directamente del usuario. Tomemos, por ejemplo, el script del capítulo anterior:

```
#!/bin/bash
# test-integer2: evaluate the value of an integer.
INT=-5
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
if [ $INT -eq 0 ]; then
echo "INT is zero."
else
if [ $INT -lt 0 ]; then
echo "INT is negative."
else
echo "INT is positive."
fi
if [ $((INT % 2)) -eq 0 ]; then
echo "INT is even."
else
echo "INT is odd."
fi
fi
else
echo "INT is not an integer." >&2
exit 1
fi
```

Cada vez que queremos cambiar el valor de `INT`, tenemos que editar el script. Sería mucho más útil si el script pudiera pedir un valor al usuario. En este capítulo, empezaremos a ver como podemos añadir interactividad a nuestros programas.

read - Lee valores de la entrada estándar

El comando incorporado `read` se usa para leer una única línea de la entrada estándar. Este comando puede usarse para leer entrada de teclado o, cuando se emplea redirección, una línea de datos desde un archivo. El comando tiene la siguiente sintaxis:

```
read [-opciones] [variable...]
```

donde *opciones* es una o más opciones disponibles listadas a continuación y *variable* es el nombre de una o más variables para almacenar el valor de entrada. Si no se proporciona ningún nombre de variable, la variable de shell `REPLY` contiene la línea de datos.

Básicamente, `read` asigna campos desde la entrada estándar hacia las variables especificadas. Si modificamos nuestro script de evaluación de entero para usar `read`, aparecería así:

```
#!/bin/bash
# read-integer: evaluate the value of an integer.
echo -n "Please enter an integer -> "
read int
```

```
if [[ "$int" =~ ^-?[0-9]+$ ]]; then
if [ $int -eq 0 ]; then
echo "$int is zero."
else
if [ $int -lt 0 ]; then
echo "$int is negative."
else
echo "$int is positive."
fi
if [ $((int % 2)) -eq 0 ]; then
echo "$int is even."
else
echo "$int is odd."
fi
fi
else
echo "Input value is not an integer." >&2
exit 1
fi
```

Usamos `echo` con la opción `-n` (que elimina la nueva línea en blanco en la salida) para mostrar un prompt, y luego usamos `read` para introducir un valor para la variable `int`. Ejecutando este script obtenemos esto:

```
[me@linuxbox ~]$ read-integer
Please enter an integer -> 5
5 is positive.
5 is odd.
```

`read` puede asignar entrada a múltiples variables, como se muestra en este script:

```
#!/bin/bash
# read-multiple: read multiple values from keyboard
echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5
echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

En este script, asignamos y mostramos los cinco valores. Fíjate como se comporta `read` cuando se le dan diferentes números de valores:

```
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e'
[me@linuxbox ~]$ read-multiple
Enter one or more values > a
```

```
var1 = 'a'
var2 = ''
var3 = ''
var4 = ''
var5 = ''
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e f g
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e f g'
```

Si `read` recibe menos que el número esperado, las variables extra estarán vacías, mientras que una cantidad excesiva de entrada provoca que la última variable contenga toda la entrada extra.

Si no se listan variables tras el comando `read`, se le asignará a una variable de shell, `REPLY`, toda la entrada:

```
#!/bin/bash
# read-single: read multiple values into default variable
echo -n "Enter one or more values > "
read
echo "REPLY = '$REPLY'"
```

Ejecutar este script produce esto:

```
[me@linuxbox ~]$ read-single
Enter one or more values > a b c d

REPLY = 'a b c d'
```

Opciones

`read` soporta las siguientes opciones:

Tabla 28-1: Opciones de read

Opción	Descripción
`-a` *array*	Asigna la entrada a *array*, comenzando con el orden cero. Veremos los arrays en el Capítulo 35.
`-d` *delimitador*	El primer carácter de la cadena *delimitador* se usa para indicar el final de la entrada, en lugar que de un carácter de nueva línea.
`-e`	Usa Readline para manejar la entrada. Esto permite editar la entrada de la misma forma que la línea de comandos.
`-i` *cadena*	Usa *cadena* como una respuesta por defecto si el usuario simplemente presiona Enter. Requiere la opción `-e`.
`-n` *num*	Lee *num* caracteres de entrada en lugar de la línea completa.

`-p` *prompt*	Muestra un prompt para la entrada usando la cadena *prompt*.
`-r`	Modo raw. No interpreta las barras invertidas como escapados.
`-s`	Modo silencioso. No envía caracteres a la pantalla cuando son escritas. Es útil cuando introducimos contraseñas u otra información confidencial.
`-t` *segundos*	Tiempo muerto. Finaliza la entrada tras *segundos*. `read` devuelve un estado de salida no-cero y la entrada caduca.
`-u` *fd*	Usa entrada del descriptor de archivo *fd*, en lugar de la entrada estándar.

Usando varias opciones, podemos hacer cosas interesantes con `read`. Por ejemplo, con la opción `-p`, podemos proporcionar una cadena de prompt:

```
#!/bin/bash
# read-single: read multiple values into default variable
read -p "Enter one or more values > "
echo "REPLY = '$REPLY'"
```

Con las opciones `-t` y `-s` podemos escribir un script que lee entrada "secreta" y caduca si la salida no se completa en un tiempo especificado:

```
#!/bin/bash
# read-secret: input a secret passphrase
if read -t 10 -sp "Enter secret passphrase > " secret_pass; then
echo -e "\nSecret passphrase = '$secret_pass'"
else
echo -e "\nInput timed out" >&2
exit 1
fi
```

El script pregunta al usuario por una frase secreta y espera entrada durante 10 segundos. Si la entrada no se completa en el tiempo especificado, el script sale con un error. Como la opción `-s` no se incluye, los caracteres de la contraseña no se muestran en la pantalla al escribirla.

Es posible proporcionar al usuario una respuesta por defecto usando las opciones `-e` y `-i` juntas:

```
#!/bin/bash
# read-default: supply a default value if user presses Enter key.
read -e -p "What is your user name? " -i $USER
echo "You answered: '$REPLY'"
```

En este script, pedimos al usuario que introduzca su nombre de usuario y la variable de entorno `USER` proporciona un valor por defecto. Cuando se ejecuta el script muestra la cadena por defecto si el usuario pulsa simplemente la tecla Intro, `read` asignará la cadena por defecto a la variable `REPLY`.

```
[me@linuxbox ~]$ read-default
What is your user name? me

You answered: 'me'
```

IFS

Normalmente, el shell realiza división de palabras en la entrada proporcionada por `read`. Como hemos visto, esto significa que varias palabras separadas por uno o más espacios se convierten en elementos separados en la línea de entrada, y `read` las asigna a variables separadas. Este comportamiento lo configura una variable de shell llamada `IFS` (Internal Field Separator - Separador Interno de Campos). El valor por defecto de `IFS` es un espacio, un tabulador o un carácter de nueva línea, cada uno de los cuales separará unos elementos de otros.

Podemos ajustar el valor de `IFS` para controlar la separación de campos de entrada a `read`. Por ejemplo, el archivo `/etc/passwd` contiene líneas de datos que usan los dos puntos como separador de campos. Cambiando el valor de `IFS` a los dos puntos, podemos usar `read` par introducir contenido de `/etc/passwd` y separar campos correctamente en diferentes variables. Aquí tenemos un script que lo hace:

```
#!/bin/bash
# read-ifs: read fields from a file
FILE=/etc/passwd
read -p "Enter a username > " user_name
file_info=$(grep "^$user_name:" $FILE)
if [ -n "$file_info" ]; then
IFS=":" read user pw uid gid name home shell <<< "$file_info"
echo "User = '$user'"
echo "UID = '$uid'"
echo "GID = '$gid'"
echo "Full Name = '$name'"
echo "Home Dir. = '$home'"
echo "Shell = '$shell'"
else
echo "No such user '$user_name'" >&2
exit 1
fi
```

Este script pide al usuario que introduzca el nombre de usuario y una cuenta del sistema, luego muestra los diferentes campos encontrados en el registro del usuario del archivo `/etc/passwd`. El script contiene dos líneas interesantes. La primera es:

```
file_info=$(grep "^$user_name:" $FILE)
```

Esta línea asigna el resultado del comando `grep` a la variable `file_info`. La expresión regular usada por `grep` garantiza que el nombre de usuario sólo aparece en una línea en el archivo `/etc/passwd`.

La segunda línea interesante es esta:

```
IFS=":" read user pw uid gid name home shell <<< "$file_info"
```

La línea consta de tres partes: una asignación de variable, un comando `read` con una lista de nombres de variables como argumentos, y un nuevo y extraño operador de redirección. Veremos la asignación de variables primero.

El shell permite que una o más asignaciones de variables se produzcan inmediatamente antes de un

comando. Estas asignaciones alteran el entorno para el comando que les sigue. El efecto de la asignación es temporal; sólo cambian el entorno por la duración del comando. En nuestro caso, el valor de `IFS` se cambia al carácter dos puntos. Alternativamente, podríamos haber escrito el código así:

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

donde almacenamos el valor de `IFS`, asignamos un nuevo valor, realizamos el comando `read` y luego restauramos `IFS` a su valor original. Claramente, colocar la asignación de variable delante del comando es una forma más concisa de hacer lo mismo.

El operador `<<<` indica una *cadena-aquí*. Una cadena-aquí es como un documento-aquí, sólo que más corto, consistente en una única cadena. En nuestro ejemplo, la línea de datos del archivo `/etc/passwd` a la entrada estándar del comando `read`. Podríamos preguntarnos porque eleginmos este modo tan indirecto en lugar de:

```
echo "$file_info" | IFS=":" read user pw uid gid name home shell
```

Bueno, hay una razón...

No puedes entubar a read

Aunque el comando `read` toma normalmente entrada de la entrada estándar, no puedes hacer ésto:

```
echo "foo" | read
```

Esperaríamos que esto funcione, pero no lo hace. El comando parecerá funcionar pero la variable `REPLY` estará siempre vacía ¿Y esto por qué?

La explicación tiene que ver con la forma en que el shell maneja los entubados. En bash (y otros shells como `sh`), los entubados crean *subshells*. Estos son copias del shell y su entorno que se usan para ejecutar el comando en el entubado. En nuestro ejemplo anterior, `read` se ejecuta en un subshell.

Los subshells en sistemas como-Unix crean copias del entorno para que los procesos lo usen mientras se ejecuten. Cuando el proceso termina, la copia del entorno es destruida. Esto significa que *un subshell nunca puede alterar el entorno de su proceso padre*. `read` asigna variables, que luego pasan a ser parte del entorno. En el ejemplo anterior, `read` asigna el valor "foo" a la variable `REPLY` en el entorno de su subshell, pero cuando el comando finaliza, el subshell y su entorno son destruidos, y el efecto de la asignación se pierde.

Usar cadenas-aquí es una forma alternativa a este comportamiento. Veremos otro método en el Capítulo 36.

Validando la entrada

Con nuestra nueva habilidad para tener entrada de teclado aparece un nuevo desafío de

programación, validar la entrada. Muy a menudo la diferencia entre un programa bien escrito y uno mal escrito reside en la capacidad del programa de tratar lo inesperado. Frecuentemente, lo inesperado aparece en una mala entrada. Hemos hecho algo de esto con nuestros programas de evaluación en el capítulo anterior, donde comprobamos los valores de enteros y descartamos valores vacíos y caracteres no numéricos. Es importante realizar este tipo de comprobaciones de programación cada vez que un programa recibe entrada, para protegernos de datos no válidos. Esto es especialmente importante para programas que son compartidos por múltiples usuarios. Omitir estos salvavidas en interés de la economía podría excusarse si un programa es para que lo use sólo el autor para realizar tareas especiales. Incluso así, si el programa realiza tareas peligrosas como borrar archivos, sería prudente incluir validación de datos, por si acaso.

Aquí tenemos un programa de ejemplo que valida varios tipos de entrada:

```
#!/bin/bash
# read-validate: validate input
invalid_input () {
echo "Invalid input '$REPLY'" >&2
exit 1
}
read -p "Enter a single item > "
# input is empty (invalid)
[[ -z $REPLY ]] && invalid_input
# input is multiple items (invalid)
(( $(echo $REPLY | wc -w) > 1 )) && invalid_input
# is input a valid filename?
if [[ $REPLY =~ ^[-[:alnum:]\._]+$ ]]; then
echo "'$REPLY' is a valid filename."
if [[ -e $REPLY ]]; then
echo "And file '$REPLY' exists."
else
echo "However, file '$REPLY' does not exist."
fi
# is input a floating point number?
if [[ $REPLY =~ ^-?[[:digit:]]*\.[[:digit:]]+$ ]]; then
echo "'$REPLY' is a floating point number."
else
echo "'$REPLY' is not a floating point number."
fi
# is input an integer?
if [[ $REPLY =~ ^-?[[:digit:]]+$ ]]; then
echo "'$REPLY' is an integer."
else
echo "'$REPLY' is not an integer."
fi
else
echo "The string '$REPLY' is not a valid filename."
fi
```

Este script pide al usuario que introduzca un elemento. El elemento es analizado posteriormente para determinar su contenido. Como podemos ver, el script hace uso de muchos de los conceptos que hemos visto hasta ahora, incluidas las funciones de shell, `[[ ]]`, `(( ))`, los operadores de control &&, e `if`, así como una buena dosis de expresiones regulares.

Menús

Un tipo común de interactividad se llama *basada en menús*. En los programas basados en menús, se le presenta al usuario un lista de opciones y se le pide que elija una. Por ejemplo, podríamos imaginar un programa que presente lo siguiente:

```
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
Enter selection [0-3] >
```

Usando lo que hemos aprendido hasta ahora de escribir nuestro programa `sys_info_page`, podemos construir un programa basado en menús para realizar las tareas del menú anterior:

```
#!/bin/bash
# read-menu: a menu driven system information program
clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "
if [[ $REPLY =~ ^[0-3]$ ]]; then
if [[ $REPLY == 0 ]]; then
echo "Program terminated."
exit
fi
if [[ $REPLY == 1 ]]; then
echo "Hostname: $HOSTNAME"
uptime
exit
fi
if [[ $REPLY == 2 ]]; then
df -h
exit
fi
if [[ $REPLY == 3 ]]; then
if [[ $(id -u) -eq 0 ]]; then
echo "Home Space Utilization (All Users)"
du -sh /home/*
else
echo "Home Space Utilization ($USER)"
du -sh $HOME
fi
exit
fi
else
echo "Invalid entry." >&2
```

```
exit 1
fi
```

El script está dividido lógicamente en dos partes. La primera parte muestra el menú y acepta la respuesta del usuario. La segunda parte identifica la respuesta y lleva a cabo la acción seleccionada. Fíjate en el uso del comando `exit` en este script. Se usa aquí para prevenir que el script ejecute código innecesario después de que se haya realizado una acción. La presencia de múltiples puntos de salida en el programa es una mala idea en general (hace que la lógica del programa sea más difícil de comprender), pero funciona en este script.

Resumiendo

En este capítulo, hemos dado nuestros primeros pasos hacia la interactividad; permitiendo a los usuarios introducir datos en los programas a través del teclado. Usando las técnicas presentadas hasta ahora, es posible escribir muchos programas útiles, como programas de cálculo especializados e interfaces fáciles de usar para comandos arcaicos de la línea de comandos. En el siguiente capítulo, construiremos un concepto de programa basado en menús para hacerlo aún mejor.

Crédito extra

Es importante estudiar los programas de este capítulo cuidadosamente y tener un entendimiento completo de la forma en que están estructurados lógicamente, ya que los programas que vendrán incrementarán la complejidad. Como ejercicio, reescribe los programas de este capítulo usando el comando `test` en lugar del comando compuesto `[[ ]]`. Pista: Usa `grep` para evaluar las expresiones regulares y el estado de salida. Será una buena costumbre.

Para saber más

- El *Manual de Referencia de Bash* tiene un capítulo con los comandos incluidos, que incluye el comando `read`:
  http://www.gnu.org/software/bash/manual/bashref.html#Bash-Builtins

Control de flujo: Bucles con while / until

En el capítulo anterior, hemos desarrollado un programa basado en menús para producir varios tipos de información del sistema. El programa funciona, pero todavía tiene un problema de usabilidad significativo. Sólo ejecuta una única opción y luego termina. Peor aún, si se hace una selección no válida, el programa termina con un error, sin dar al usuario una oportunidad de probar de nuevo. Sería mejor si pudiéramos, de alguna forma, construir el programa de forma que pueda repetir la pantalla del menú y selección una y otra vez, hasta que el usuario elija salir del programa.

En este capítulo, veremos un concepto de programación llamado *bucle*, que puede usarse para hacer que partes de programa se repitan. El shell proporciona tres comandos compuestos para bucles. Veremos dos de ellos en este capítulo, y el tercero en un capítulo posterior.

Bucles

La vida diaria está llena de actividades repetitivas. Ir al trabajo cada día, pasear al perro, cortar zanahorias y todas las tareas que implican repetir una serie de pasos. Consideremos cortar una zanahoria en rodajas. Si expresamos esta actividad en pseudocódigo, sería algo parecido a esto:

1. coger una tabla de cortar

2. coger un cuchillo
3. poner la zanahoria en la tabla de cortar
4. levantar el cuchillo
5. mover la zanahoria
6. cortar la zanahoria
7. si toda la zanahoria está cortada, parar, si no, volver al paso 4

Los pasos del 4 al 7 forman un *bucle*. Las acciones dentro del bucle se repiten hasta que se alcanza la condición, "toda la zanahoria está cortada".

while

bash puede expresar una idea similar. Digamos que queremos mostrar cinco números en orden secuencial desde el uno al cinco. Un script de bash podría construirse de la siguiente forma:

```
#!/bin/bash
# while-count: display a series of numbers
count=1
while [[ $count -le 5 ]]; do
echo $count
count=$((count + 1))
done
echo "Finished."
```

Cuando se ejecuta, este script muestra lo siguiente:

```
[me@linuxbox ~]$ while-count
1
23
45
Finished.
```

La sintaxis del comando while es:

```
while comandos; do comandos; done
```

Al igual que if, while evalúa el estado de la salida de una lista de comandos. Mientras que el estado de la salida sea cero, ejecuta los comandos dentro del bucle. En el script anterior, se crea la variable count y se le asigna un valor inicial de 1. El comando while evalúa el estado de salida el comando test. Mientras el comando test devuelva un estado de salida cero, los comandos dentro del bucle se ejecutan. Al final de cada ciclo, se repite el comando test. Tras seis iteraciones del bucle, el valor de count se ha incrementado hasta 6, el comando test ya no devuelve un estado de salida de cero y el bucle termina. El programa continua con la siguiente línea a continuación del bucle.

Podemos usar un *bucle while* para mejorar el programa read-menu del capítulo anterior:

```
#!/bin/bash
# while-menu: a menu driven system information program
DELAY=3 # Number of seconds to display results
while [[ $REPLY != 0 ]]; do
clear
```

```
cat <<- _EOF_
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
_EOF_
read -p "Enter selection [0-3] > "
if [[ $REPLY =~ ^[0-3]$ ]]; then
if [[ $REPLY == 1 ]]; then
echo "Hostname: $HOSTNAME"
uptime
sleep $DELAY
fi
if [[ $REPLY == 2 ]]; then
df -h
sleep $DELAY
fi
if [[ $REPLY == 3 ]]; then
if [[ $(id -u) -eq 0 ]]; then
echo "Home Space Utilization (All Users)"
du -sh /home/*
else
echo "Home Space Utilization ($USER)"
du -sh $HOME
fi
sleep $DELAY
fi
else
echo "Invalid entry."
sleep $DELAY
fi
done
echo "Program terminated."
```

Englobando el menú en un bucle while, podemos hacer que el programa repita la pantalla de menú tras cada selección. El bucle continua mientras `que REPLY` no sea igual a "0" y el menú se muestra de nuevo, dando al usuario la oportunidad de realizar otra selección. Al final de cada acción, se ejecuta un comando `sleep` de forma que el programa espera unos segundos para permitir que el resultado de la selección se vea antes de borrar la pantalla y volver a mostrar el menú. Una vez que `REPLY` es igual a "0", indicando la selección "quit", termina el bucle y continua la ejecución con la siguiente línea `done`.

Salir de un bucle

`bash` proporciona dos comandos internos que pueden usarse para controlar el flujo del programa dentro de los bucles. El comando `break` termina un bucle inmediatamente, y el control del programa sigue con la siguiente sentencia que siga al bucle. El comando `continue` hace que se salte el resto del bucle, y el control del programa sigue con la siguiente iteración del bucle. Aquí vemos una versión del programa `while-menu` incorporando tanto `break` como `continue`:

```
#!/bin/bash
# while-menu2: a menu driven system information program
DELAY=3 # Number of seconds to display results
while true; do
clear
cat <<- _EOF_
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
_EOF_
read -p "Enter selection [0-3] > "
if [[ $REPLY =~ ^[0-3]$ ]]; then
if [[ $REPLY == 1 ]]; then
echo "Hostname: $HOSTNAME"
uptime
sleep $DELAY
continue
fi
if [[ $REPLY == 2 ]]; then
df -h
sleep $DELAY
continue
fi
if [[ $REPLY == 3 ]]; then
if [[ $(id -u) -eq 0 ]]; then
echo "Home Space Utilization (All Users)"
du -sh /home/*
else
echo "Home Space Utilization ($USER)"
du -sh $HOME
fi
sleep $DELAY
continue
fi
if [[ $REPLY == 0 ]]; then
break
fi
else
echo "Invalid entry."
sleep $DELAY
fi
done
echo "Program terminated."
```

En esta versión del script, configuramos un *bucle sin fin* (uno que nunca termina por sí sólo) usando el comando `true` para proporcionar un estado de salida a `while`. Como `true` siempre sale con un estado de salida de cero, el bucle nunca terminará. Esta es sorprendentemente una práctica de programación común. Como el bucle no termina por sí sólo, corresponde al programador proporcionar algún tipo de interrupción del bucle cuando haga falta. En este script, el comando `break` se usa para salir del bucle cuando se elige la selección "0". El comando `continue` se ha

incluido al final de las otras opciones del script para ofrecer una ejecución más eficiente. Usando `continue`, el script se saltará código que no se necesita cuando se identifica una selección. Por ejemplo, si se elige la selección "1" y esta se identifica, no hay razón para probrar el resto de selecciones.

El comando `until` es casi como `while`, excepto que en lugar de salir de un bucle cuando se encuentra un estado de salida cero, hace lo contrario. Un *bucle until* continua hasta que reciba un estado de salida cero. En nuestro script `while-count`, continuamos el bucle mientras el valor de la variable `count` sea menor o igual a 5. Podríamos tener el mismo resultado codificando el script con `until`:

```
#!/bin/bash
# until-count: display a series of numbers
count=1
until [[ $count -gt 5 ]]; do
echo $count
count=$((count + 1))
done
echo "Finished."
```

Cambiando la expresión test a `$count -gt 5, until`, terminaremos el bucle en el momento correcto. La decisión de si el uso del bucle `while` o `until` es normalmente una forma de elegir el que permite escribir el test más claro. Leyendo archivos con bucles
`while` y `until` pueden procesar entrada estándar. Esto permite que los archivos sean procesados con bucles while y until. En el siguiente ejemplo, mostraremos el contenido del archivo `distros.txt` usado en los capítulos anteriores:

```
#!/bin/bash
# while-read: read lines from a file
while read distro version release; do
printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
$distro \
$version \
$release
done < distros.txt
```

Para redireccionar una archivo al bucle, colocamos el operador de redirección tras la sentencia `done`. El bucle usara `read` para introducir los campos desde el archivo redirigido. El comando `read` saldrá cada vez que lea una línea, con un estado de salida cero hasta que se alcance el final del archivo. En este punto, saldrá con un estado de salido no-cero, terminando de esta forma el loop. También es posible entubar la entrada estándar dentro de un bucle:

```
#!/bin/bash
# while-read2: read lines from a file
sort -k 1,1 -k 2n distros.txt | while read distro version release;
do
printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
$distro \
$version \
$release
done
```

Aquí hemos tomado la salida del comando `sort` y hemos mostrado la cadena de texto. Sin embargo es importante recordar que como un entubado ejecutará el bucle en un subshell, cualquier variable creada o asignada dentro del bucle se perderá cuando termine el bucle.

Resumiendo

Con la introducción de los bucles, y nuestros anteriores encuentros con las ramificaciones, las subrutinas y secuencias, hemos visto la mayoría de los tipos de control de flujo usados en programas. `bash` se guarda algunos trucos en la manga, pero son refinamientos de estos conceptos básicos.

Para saber más

- La *Guía de Bash para Principiantes* del Proyecto de Documentación Linux tiene algunos ejemplos más de bucles con while:
  http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_09_02.html
- La Wikipedia tiene un artículo sobre bucles, que es parte de un artículo mayor sobre control de flujo:
  http://en.wikipedia.org/wiki/Control_flow#Loops

Solución de Problemas

A medida que nuestros scripts se hacen más complejos, es hora de echar un vistazo a lo que ocurre cuando las cosas no funcionan y no hacen lo que queremos. En este capítulo, veremos algunos tipos comunes de errores que ocurren en nuestros scripts, y describiremos unas pocas técnicas útiles que pueden usarse para localizar y erradicar los problemas.

Errores sintácticos

Un tipo común de error es el *sintáctico*. Los errores sintácticos implica errores de escritura en algunos elementos de la sintaxis del shell. En la mayoría de los casos, estos tipos de errores liderarán los rechazos del shell a ejecutar el script.

En el siguiente párrafo, usaremos este script para comprobar los tipos comunes de errores:

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
if [ $number = 1 ]; then
echo "Number is equal to 1."
else
echo "Number is not equal to 1."
fi
```

Tal como está escrito, este script se ejecuta con éxito:

```
[me@linuxbox ~]$ trouble

Number is equal to 1.
```

Comillas perdidas

Si editamos nuestro script y eliminamos las comillas finales del argumento que sigue al primer comando `echo`:

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
if [ $number = 1 ]; then
echo "Number is equal to 1.
else
echo "Number is not equal to 1."
fi
```

y vemos que ocurre:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 10: unexpected EOF while looking for
matching `"'
/home/me/bin/trouble: line 13: syntax error: unexpected end of
file
```

Genera dos errores. Interesantemente, los números de línea reportados no son de donde se han eliminado las comillas, si no que están mucho más avanzado el programa. Podemos ver por qué, si seguimos el programa tras las comillas perdidas. `bash` continua buscando las comillas de cierre hasta que encuentra unas, lo que ocurre inmediatamente tras el segundo comando `echo`. `bash` se queda muy confundido tras esto, y la sintaxis del comando `if` se rompe porque la sentencia `fi` está ahora dentro de una cadena entrecomillada (pero abierta).

En scripts largos, este tipo de errores pueden ser algo difíciles de encontrar. Usar un editor con resaltado sintáctico ayuda. Si está instalada la versión completa de `vim`, puede activarse el resaltado sintáctico introduciendo el comando:

```
:syntax on
```

Símbolos perdidos o inesperados

Otro error común es olvidarse de completar un comando compuesto, como `if` o `while`. Veamos que ocurre si eliminamos el punto y coma tras el `test` en el comando `if`:

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
if [ $number = 1 ] then
echo "Number is equal to 1."
else
echo "Number is not equal to 1."
fi
```

El resultado es:

```
[me@linuxbox ~]$ trouble
```

```
/home/me/bin/trouble: line 9: syntax error near unexpected token
`else'
/home/me/bin/trouble: line 9: `else'
```

De nuevo, el mensaje de error apunta a un error que ocurre más tarde que el problema real. Lo que ocurre es realmente interesante. Como dijimos, `if` acepta una lista de comandos y evalúa el código de salida del último comando de la lista. En nuestro programa, nuestra intención es que esta lista sea un único comando, `[`, un sinónimo de test. El comando `[` toma lo que le sigue como una lista de argumentos; en nuestro caso, cuatro argumentos: `$number`, `1`, `=`, y `]`. Con el punto y coma eliminado, se añade la palabra `then` a la lista de argumentos, lo que es sintácticamente legal. El siguiente comando echo es legal, también. Se interpreta como otro comando en la lista de comandos que `if` evaluará como código de salida. A continuación encontramos else, pero está fuera de sitio, ya que el shell lo reconoce como una *palabra reservada* (una palabra que tiene un significado especial para el shell) y no el nombre de un comando, de ahí el mensaje de error.

Expansiones inesperadas

Es posible tener errores que sólo ocurren de forma intermitente en un script. A veces el script se ejecutará correctamente y otras veces fallará debido al resultado de una expansión. Si devolvemos nuestro punto y coma perdido y cambiamos el valor de `number` a una variable vacía, podemos comprobarlo:

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=
if [ $number = 1 ]; then
echo "Number is equal to 1."
else
echo "Number is not equal to 1."
fi
```

Ejecutar el script con este cambio da como resultado esta salida:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 7: [: =: unary operator expected
Number is not equal to 1.
```

Obtenemos un mensaje de error bastante críptico, seguido de la salida del segundo comando `echo`. El problema es la expansión de la variable `number` dentro del comando `test`. Cuando el comando:

```
[ $number = 1 ]
```

se somete a la expansión con `number` estando vacío, el resultado es este:

```
[ = 1 ]
```

que no es válido y se genera el error. El operador = es un operador binario (requiere un valor a cada lado), pero el primer valor no está, así que el comando `test` espera un operador unario (como `-Z`) en su lugar. Más adelante, como el `test` ha fallado (debido al error), el comando `if` recibe un código de salida no-cero y actual de acuerdo con esto, y se ejecuta el segundo comando `echo`.

Este problema puede corregirse añadiendo comillas alrededor del primer argumento en el comando `test`:

```
[ "$number" = 1 ]
```

De esta forma cuando se produce la expansión, el resultado será este:

```
[ "" = 1 ]
```

que proporciona el número correcto de argumentos. Además para las cadenas vacías, las comillas deben usarse en casos donde un valor puede expandirse en cadenas multipalabra, como nombres de archivo que contengan espacios incluidos.

Errores lógicos

Al contrario de los errores sintácticos, los *errores lógicos* no impiden que se ejecute un script. El script se ejecutará, pero no producirá el resultado deseado, debido a un problema con su lógica. Hay un número incontable de errores lógicos posibles, pero aquí os dejo unos pocos de los tipos más comunes que se encuentran en scripts:

1. **Expresiones condicionales incorrectas.** Es fácil codificar mal un if/then/else y llevar a cabo una lógica equivocada. A veces la lógica se invierte, o queda incompleta.
2. **Errores "Fuera por uno".** Cuando codificamos bucles que utilizan contadores, es posible pasar por alto que el bucle requiere que el contador empiece por cero en lugar de uno, para que concluya en el punto correcto. Este tipo de errores producen un bucle que "va más allá del final" contando demasiado lejos, o que pierde la última iteración del bucle terminando una iteración antes de tiempo.
3. **Situaciones no previstas.** La mayoría de los errores lógicos son producidos porque un programa se encuentra con datos o situaciones que no han sido previstas por el programador. Esto puede incluir tanto expansiones no previstas, como un nombre de archivo que contiene espacios y que se expande en múltiples argumentos de comando en lugar de un nombre de archivo único.

Programación defensiva

Es importante verificar las suposiciones cuando programamos. Esto significa una evaluación cuidadosa del estado de salida de programas y comandos que se usan en un script. Aquí hay un ejemplo, basado en hechos reales. Un desafortunado administrador de sistemas escribió un script para realizar una tarea de mantenimiento en un importante servidor. El script contenía las siguientes dos líneas de código:

```
cd $dir_name
rm *
```

No hay nada intrínsecamente malo en estas dos líneas, ya que el directorio citado en la variable, `dir_name` existe. Pero ¿que ocurre si no es así? En ese caso, el comando `cd` falla y el script continua con al siguiente línea y borra los archivos en el directorio de trabajo actual. ¡No es para nada el resultado deseado! Este desafortunado administrador destruyó una parte importante del servidor por esta decisión de diseño.

Veamos algunas formas en que este diseño puede mejorarse. Primero, sería prudente hacer que la

ejecución de `rm` dependea del éxito de `cd`:

```
cd $dir_name && rm *
```

De esta forma, si el comando `cd` falla, el comando `rm` no se ejecuta. Esto es mejor, pero aún deja abierta la posibilidad de que la variable, `dir_name`, esté sin configurar o vacía, lo que resultaría en que los archivos en el directorio home del usuario se borrarían. Esto podría evitarse también comprobando si `dir_name` en realidad contiene el nombre de un directorio existente:

```
[[ -d $dir_name ]] && cd $dir_name && rm *
```

A menudo, es mejor terminar el script con un error cuando ocurre una situación como la anterior:

```
# Delete files in directory $dir_name
if [[ ! -d "$dir_name" ]]; then
echo "No such directory: '$dir_name'" >&2
exit 1
fi
if ! cd $dir_name; then
echo "Cannot cd to '$dir_name'" >&2
exit 1
fi
if ! rm *; then
echo "File deletion failed. Check results" >&2
exit 1
fi
```

Aquí, comprobamos tanto el nombre, para ver si es el de un directorio existente, como el éxito del comando `cd`. Si ambos fallan, se envía un error descriptivo al error estándar y el script termina con un estado de salida de uno para indicar un fallo.

Verificando la entrada

Una regla general de buena programación es que, si un programa acepta entrada, debe ser capaz de gestionar cualquier cosa que reciba. Esto normalmente significa que la entrada debe ser cuidadosamente filtrada, para asegurar que sólo se acepte entrada válida para su procesamiento posterior. Vimos un ejemplo de esto en nuestro capítulo anterior cuando estudiamos el comando `read`. Un script conteniendo el siguiente test para verificar una selección de menú:

```
[[ $REPLY =~ ^[0-3]$ ]]
```

Este test es muy específico. Sólo devolverá un estado de salida cero y la cadena devuelta por el usuario es un número en el rango entre cero y tres. Nada más será aceptado. A veces estos tipos de tests pueden ser muy complicados de escribir, pero el esfuerzo es necesario para producir un script de alta calidad.

El diseño va en función del tiempo

Cuando estudiaba diseño industrial en el instituto, un sabio profesor expuso que el grado de diseño en un proyecto viene determinado por la cantidad de tiempo que le den al diseñador. Si te dieran

cinco minutos para diseñar un dispositivo "que mate moscas," diseñarías un matamoscas. Si te dieran cinco meses, volverías con un "sistema anti-moscas" guiado por láser.

El mismo principio se aplica a la programación. A veces un script "rápido y sucio" servirá si sólo va a usarse una vez y sólo va a usarlo el programador. Este tipo de script es común y debería desarrollarse rápidamente para hacer que el esfuerzo sea rentable. Tales scripts no necesitan muchos comentarios ni comprobaciones defensivas. En el otro extremo, si el script está destinado a un *uso en producción*, o sea, un script que se usará una y otra vez para una tarea importante o por muchos usuarios, necesita un desarrollo mucho más cuidadoso.

Testeo

El testeo es un paso importante en todo tipo de desarrollo de software, incluidos los scripts. Hay un dicho en el mundo del software libre, "libera pronto, libera a menudo", que refleja este hecho. Liberando pronto y a menudo, el software se hace más expuesto al uso y al testeo. La experiencia nos ha demostrado que los bugs son más fáciles de encontrar, y mucho menos caros de arreglar, si se encuentran pronto en el ciclo de desarrollo.

En un tema anterior, vimos cómo podemos usar stubs para verificar el flujo del programa. Desde las primeras fases del desarrollo del script, son una técnica valiosa para comprobar el progreso de nuestro trabajo.

Echemos un vistazo al problema de eliminación de archivos anterior y veamos como puede codificarse para que sea más fácil el testeo. Testear el fragmento original de código puede ser peligroso, ya que su propósito es borrar archivos, pero podríamos modificar el código para hacer que el testeo sea seguro:

```
if [[ -d $dir_name ]]; then
if cd $dir_name; then
echo rm * # TESTING
else
echo "cannot cd to '$dir_name'" >&2
exit 1
fi
else
echo "no such directory: '$dir_name'" >&2
exit 1
fi
exit # TESTING
```

Como las condiciones del error ya muestran mensajes útiles, no tenemos que añadir anda. El cambio más importante es colocar un comando `echo` justo antes del comando `rm` para permitir al comando y su lista de argumentos expandidos que se muestren, en lugar de que el comando se ejecute en realidad. Este cambio permite la ejecución segura del código. Al final del fragmento de código, colocamos un comando `exit` para concluir el test y prevenir que cualquier otra parte del script se ejecute. La necesidad de esto variará segun el diseño del script.

También incluimos algunos comentarios que actúan como "marcadores" de nuestros cambios referidos al testeo. Estos pueden usarse para ayudarnos a encontrar y eliminar los cambios cuando esté completo el testeo.

Casos de testeo

Para realizar un testeo útil, es importante desarrollar y aplicar buenos *casos de testeo*. Esto se hace eligiendo cuidadosamente la entrada de datos o las condiciones de operación que reflejen casos *límite*. En nuestro fragmento de código (que es muy simple), queremos saber como se comporta el código bajo tres condiciones específicas:

1. `dir_name` contiene el nombre de un directorio existente
2. `dir_name` contiene el nombre de un directorio no existente
3. `dir_name` está vacío

Realizando el testeo con cada una de estas condiciones, obtenemos una buena *cobertura de testeo*.

Tal como con el diseño, el testo depende el tiempo, también. No todas las partes del script necesitan ser testeadas intensivamente. De hecho es una forma de determinar que es lo más importante. Como podría ser potencialmente destructivo si no funciona bien, nuestro fragmento de código se merece una consideración cuidadosa tanto durante el diseño como el testeo.

Depuración

Si el testeo revela un problema con un script, el siguiente paso es la depuración. "Un problema" normalmente significa que el script, de alguna forma, no está realizando lo que espera el programador. Si este es el caso, necesitamos determinar cuidadosamente que es lo que el script está haciendo en realidad y por qué. Encontrar errores puede implicar, a veces, mucho trabajo de detective.

Un script bien diseñado debería ayudar. Debería estar programado defensivamente, para detectar condiciones anormales y proporcionar retroalimentación útil al usuario. A veces, sin embargo, los problemas son algo extraños e inesperados, y se requieren más técnicas involucradas.

Encontrando el área del problema

En algunos scripts, particularmente los largos, es útil a veces aislar el área del script relacionado con el problema. Esto no siempre será el error real, pero aislarlo a menudo proporciona un mejor enfoque del problema real. Una técnica que puede usarse para aislar código es "comentar" secciones de un script. Por ejemplo, nuestro fragmento borrador de archivos podría modificarse para determinar si la sección eliminada está relacionada con un error:

```
if [[ -d $dir_name ]]; then
if cd $dir_name; then
rm *
else
echo "cannot cd to '$dir_name'" >&2
exit 1
fi
# else
# echo "no such directory: '$dir_name'" >&2
# exit 1
fi
```

Colocar símbolos de comentarios al principio de cada línea en una sección lógica del script, impide que la sección se ejecute. El testeo puede ahora ser realizado de nuevo, para ver si la eliminación

del código tiene algún impacto en el comportamiento del error.

Trazado

Los errores son a menudo casos de un flujo lógico inesperado dentro de un script. O sea, partes del script o nunca se ejecutan, o se ejecutan en un orden erróneo o en el momento equivocado. Para ver el flujo actual del programa, usamos una técnica llamada *tracing* o trazado.

Un método de trazado implica colocar mensajes informativos en un script que muestre la localización de la ejecución. Podemos añadir mensajes a nuestro fragmento de código:

```
echo "preparing to delete files" >&2
if [[ -d $dir_name ]]; then
if cd $dir_name; then
echo "deleting files" >&2
rm *
else
echo "cannot cd to '$dir_name'" >&2
exit 1
fi
else
echo "no such directory: '$dir_name'" >&2
exit 1
fi
echo "file deletion complete" >&2
```

Enviamos mensajes al error estándar para separalo de la salida normal. No indentamos las líneas que contienen los mensajes, así es más fácil encontrarlas cuando sea el momento de borrarlas.

Ahora, cuando se ejecuta el script, es posible ver que el borrado de archivos se ha realizado:

```
[me@linuxbox ~]$ deletion-script
preparing to delete files
deleting files
file deletion complete
[me@linuxbox ~]$
```

`bash` también proporciona un método de trazado, implementado por la opción `-x` y el comando `set` con la opción `-x`. Usando nuestro anterior script `trouble`, podemos activar el trazado para todo el script añadiendo la opción `-x` en la primera línea:

```
#!/bin/bash -x
# trouble: script to demonstrate common errors
number=1
if [ $number = 1 ]; then
echo "Number is equal to 1."
else
echo "Number is not equal to 1."
fi
```

Cuando se ejecuta, el resultado es el siguiente:

```
[me@linuxbox ~]$ trouble
```

```
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.
```

Cuando el trazado está activado, vemos que los comandos se ejecutan con expansiones aplicadas. El signo más al principio de la línea indica el resultado del trazado para distinguirlo de las líneas de salida normal. El signo más es el carácter por defecto de la salida del trazado. Está contenido en la variable de shell `PS4` (prompt string 4 - cadena de prompt 4). El contenido de esta variable puede ajustarse para hacer el prompt más útil. Aquí, modificamos el contenido de la variable para incluir el número de línea actual en el script donde se realiza el trazado. Fíjate que se requieren comillas simples para impedir la expansión hasta que el prompt se haya usado realmente:

```
[me@linuxbox ~]$ export PS4='$LINENO + '
[me@linuxbox ~]$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

Para realizar un trazado en una porción concreta del un script, en lugar del script completo, podemos usar el comando `set` con la opción `-x`:

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
set -x # Turn on tracing
if [ $number = 1 ]; then
echo "Number is equal to 1."
else
echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

Usamos el comando `set` con la opción `-x` activada para habilitar el trazado y la opción `+x` para desactivar el trazado. Esta técnica puede usarse para examinar múltiples porciones de un script problemático.

Examinando valores durante la ejecución

A menudo es útil, junto con el trazado, mostrar el contenido de variables para ver los trabajos internos de un script mientras se está ejecutando. Normalmente el truco es aplicar instancias adicionales de `echo`:

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
echo "number=$number" # DEBUG
set -x # Turn on tracing
if [ $number = 1 ]; then
echo "Number is equal to 1."
else
```

```
echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

En este ejemplo trivial, simplemente mostramos el valor del número de la variable y marcamos la línea añadida con un comentario para facilitar su posterior identificación y eliminación. Esta técnica es particularmente útil cuando vemos el comportamiento de bucles y aritmética dentro de scripts.

Resumiendo

En este capítulo, hemos visto sólo unos pocos de los problemas que pueden surgir durante el desarrollo de un script. De acuerdo, hay muchos más. Las técnicas descritas aquí permitirán encontrar los errores más comunes. El depurado es un arte fino que puede desarrollarse a través de la experiencia, tanto en conocimiento de cómo evitar errores (probando constantemente durante el desarrollo) como encontrando errores (uso efectivo del trazado).

Para saber más

- La Wikipedia tiene un par de artículos cortos sobre errores sintácticos y lógicos:
  http://en.wikipedia.org/wiki/Syntax_error
  http://en.wikipedia.org/wiki/Logic_error
- Hay muchos recursos online sobre aspectos técnicos de la programación en `bash`:
  http://mywiki.wooledge.org/BashPitfalls
  http://tldp.org/LDP/abs/html/gotchas.html
  http://www.gnu.org/software/bash/manual/html_node/Reserved-Word-Index.html
- *Eric Raymonds's The Art of Unix Programing* (El Arte de la Programación en Unix de Eric Raymond) es un gran recurso para aprender los conceptos básicos que se encuentran en programas Unix bien escritos. Muchas de esas ideas se aplican a los scripts de shell:
  http://www.faqs.org/docs/artu/
  http://www.faqs.org/docs/artu/ch01s06.html
- Para tareas pesadas de depuración, está el Bash Debugger:
  http://bashdb.sourceforge.net/

Control de Flujo: Ramificando con case

En este capítulo, continuaremos viendo el control de flujo. En este Capítulo 28, hemos construido algunos menús simples y la lógica usada para actuar ante una elección del usuario. Para hacerlo, usamos una serie de comandos `if` para identificar cual de las posibles elecciones ha sido seleccionada. Este tipo de construcciones aparecen frecuentemente en programas, tanto es así que muchos lenguajes de programación (incluyendo el shell) proporcionan un mecanismo de control de flujo para decisiones con múltiples opciones.

case

El comando compuesto de `bash` para múltiples opciones se llama `case`. Tiene la siguiente sintaxis:

```
case palabra in
[patrón [| patrón]...) comandos ;;]...
esac
```

Si miramos el programa read-menu del Capítulo 28, vemos la lógica usada para actuar en una selección del usuario:

```
#!/bin/bash
# read-menu: a menu driven system information program
clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "
if [[ $REPLY =~ ^[0-3]$ ]]; then
if [[ $REPLY == 0 ]]; then
echo "Program terminated."
exit
fi
if [[ $REPLY == 1 ]]; then
echo "Hostname: $HOSTNAME"
uptime
exit
fi
if [[ $REPLY == 2 ]]; then
df -h
exit
fi
if [[ $REPLY == 3 ]]; then
if [[ $(id -u) -eq 0 ]]; then
echo "Home Space Utilization (All Users)"
du -sh /home/*
else
echo "Home Space Utilization ($USER)"
du -sh $HOME
fi
exit
fi
else
echo "Invalid entry." >&2
exit 1
fi
```

Usando case podemos reemplazar esta lógica con algo más simple:

```
#!/bin/bash
# case-menu: a menu driven system information program
clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
```

```
0. Quit
"
read -p "Enter selection [0-3] > "
case $REPLY in
0) echo "Program terminated."
exit
;;
1) echo "Hostname: $HOSTNAME"
uptime
;;
2) df -h
;;
3) if [[ $(id -u) -eq 0 ]]; then
echo "Home Space Utilization (All Users)"
du -sh /home/*
else
echo "Home Space Utilization ($USER)"
du -sh $HOME
fi
;;
*) echo "Invalid entry" >&2
exit 1
;;
esac
```

El comando `case` mira el valor de *palabra*, en nuestro ejemplo, el valor de la variable `REPLY`, y luego trata de compararlo con uno de los patrones especificados. Cuando encuentra una coincidencia, los *comandos* asociados con el patrón especificado se ejecutan. Una vez que se encuentra una coincidencia, no se intentan más coincidencias.

Patrones

Los patrones usados por `case` son los mismos que aquellos usados por la expansión de rutas. Los patrones terminan con un carácter ")":

Tabla 32-1: Ejemplos de patrones de case

Patrón	Descripción
`a)`	Coincide si *palabra* es igual a "a".
`[[:alpha:]])`	Coincide si *palabra* es un caracter alfabético único.
`???)`	Coincide si *palabra* tiene exactamente tres caracteres de longitud.
`*.txt)`	Coincide si *palabra* finaliza con los caracteres ".txt".
`*)`	Coincide con cualquier valor de *palabra*. Es una buena práctica incluirlo como el último patrón en un comando `case`, para tomar cualquier valor de *palabra* que no coincida con un patrón previo; o sea, para tomar todos los valores no válidos posibles.

Aquí tenemos un ejemplo de cómo funcionan los patrones:

```
#!/bin/bash
read -p "enter word > "
case $REPLY in
[[:alpha:]]) echo "is a single alphabetic character." ;;
[ABC][0-9]) echo "is A, B, or C followed by a digit." ;;
???) echo "is three characters long." ;;
*.txt) echo "is a word ending in '.txt'" ;;
*) echo "is something else." ;;
esac
```

También es posible combinar múltiples patrones usando el carácter de la barra vertical como separador. Esto crea un patrón de "o" condicional. Esto es útil para cosas como manejar caracteres tanto mayúsculas como minúsculas. Por ejemplo:

```
#!/bin/bash
# case-menu: a menu driven system information program
clear
echo "
Please Select:
A. Display System Information
B. Display Disk Space
C. Display Home Space Utilization
Q. Quit
"
read -p "Enter selection [A, B, C or Q] > "
case $REPLY in
q|Q) echo "Program terminated."
exit
;;
a|A) echo "Hostname: $HOSTNAME"
uptime
;;
b|B) df -h
;;
c|C) if [[ $(id -u) -eq 0 ]]; then
echo "Home Space Utilization (All Users)"
du -sh /home/*
else
echo "Home Space Utilization ($USER)"
du -sh $HOME
fi
;;
*) echo "Invalid entry" >&2
exit 1
;;
esac
```

Aquí, modificamos el programa `case-menu` para usar letras en lugar de dígitos para las selecciones del menú. Fíjate cómo los nuevos patrones permiten introducir tanto letras mayúsculas como minúsculas.

Realizando múltiples acciones

En versiones de bash anteriores a la 4.0, case permite que se realice una única acción en una coincidencia exitosa. Tras una coincidencia exitosa, el comando terminaría. Aquí vemos un script que prueba un carácter:

```
#!/bin/bash
# case4-1: test a character
read -n 1 -p "Type a character > "
echo
case $REPLY in
[[:upper:]]) echo "'$REPLY' is upper case." ;;
[[:lower:]]) echo "'$REPLY' is lower case." ;;
[[:alpha:]]) echo "'$REPLY' is alphabetic." ;;
[[:digit:]]) echo "'$REPLY' is a digit." ;;
[[:graph:]]) echo "'$REPLY' is a visible character." ;;
[[:punct:]]) echo "'$REPLY' is a punctuation symbol." ;;
[[:space:]]) echo "'$REPLY' is a whitespace character." ;;
[[:xdigit:]]) echo "'$REPLY' is a hexadecimal digit." ;;
esac
```

Ejecutar este script produce esto:

```
[me@linuxbox ~]$ case4-1
Type a character > a
'a' is lower case.
```

El script funciona en su mayor parte, pero falla si un carácter coincide en más de una clase de caracter POSIX. Por ejemplo, el carácter "a" es tanto mayúscula como minúscula, así como un dígito hexadecimal. En bash anterior a la versión 4.0 no hay forma de que case coincida en más de una prueba. Las versiones modernas de bash, añaden la notación ";;&" para terminar cada acción, así que podemos hacer esto:

```
#!/bin/bash
# case4-2: test a character
read -n 1 -p "Type a character > "
echo
case $REPLY in
[[:upper:]]) echo "'$REPLY' is upper case." ;;&
[[:lower:]]) echo "'$REPLY' is lower case." ;;&
[[:alpha:]]) echo "'$REPLY' is alphabetic." ;;&
[[:digit:]]) echo "'$REPLY' is a digit." ;;&
[[:graph:]]) echo "'$REPLY' is a visible character." ;;&
[[:punct:]]) echo "'$REPLY' is a punctuation symbol." ;;&
[[:space:]]) echo "'$REPLY' is a whitespace character." ;;&
[[:xdigit:]]) echo "'$REPLY' is a hexadecimal digit." ;;&
esac
```

Cuando ejecutamos este script, obtenemos esto:

```
[me@linuxbox ~]$ case4-2
Type a character > a
'a' is lower case.
```

```
'a' is alphabetic.
'a' is a visible character.
'a' is a hexadecimal digit.
```

Añadir la sintaxis ";;&" permite que case continúe con el siguiente test en lugar de simplemente terminar.

Resumiendo

El comando case es un añadido útil en nuestra maleta llena de trucos de programación. Como veremos en el próximo capítulo, es la herramienta perfecta para manejar ciertos tipos de problemas.

Para saber más

- La sección de Estructuras Condicionales del *Manual de Referencia de Bash* describe el comando case al detalle:
  http://tiswww.case.edu/php/chet/bash/bashref.html#SEC21
- La *Guía Avanzada de Scripting en Bash* ofrece más ejemplos de aplicaciones de `case`:
  http://tldp.org/LDP/abs/html/testbranch.html

Parámetros Posicionales

Una prestación que ha estado desaparecida en nuestros programas es la capacidad de aceptar y procesar opciones y argumentos de la línea de comandos. En este capítulo, examinaremos las funcionalidades del shell que permiten que nuestros programas accedan al contenido de la línea de comandos.

Accediendo a la línea de comandos

El shell ofrece una serie de variables llamadas *parámetros posicionales* que contienen las palabras individuales de la línea de comandos. Las variables se nombran del `0` al `9`. Puede comprobarse así:

```
#!/bin/bash
# posit-param: script to view command line parameters
echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

Un script muy simple que muestre el valor de las variables `$0-$9`. Cuando lo ejecutamos sin argumentos de la línea de comandos:

```
[me@linuxbox ~]$ posit-param
```

```
$0 = /home/me/bin/posit-param
$1 =
$2 =
$3 =
$4 =
$5 =
$6 =
$7 =
$8 =
$9 =
```

Incluso cuando no se le dan argumentos, `$0` siempre contendrá el primer elemento que aparezca en la línea de comandos, que es la ruta del programa que se está ejecutando. Cuando se le pasan argumentos, vemos el resultado:

```
[me@linuxbox ~]$ posit-param a b c d
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

Nota: En realidad puedes acceder a más de nueve parámetros usando la expansión de parámetros. Para especificar un número mayor que nueve, incluimos el número entre llaves. Por ejemplo ${10}, ${55}, ${211} y así sucesivamente.

Determinando el número de argumentos

El shell también ofrece una variable, `$#`, que muestra el número de argumentos en la línea de comandos:

```
#!/bin/bash
# posit-param: script to view command line parameters
echo "
Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

El resultado:

```
[me@linuxbox ~]$ posit-param a b c d
Number of arguments: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =

$9 =
```

shift - Accediendo a muchos argumentos

Pero, ¿qué ocurre cuando le damos al programa un gran número de argumentos como en el ejemplo siguiente?:

```
[me@linuxbox ~]$ posit-param *
Number of arguments: 82
$0 = /home/me/bin/posit-param
$1 = addresses.ldif
$2 = bin
$3 = bookmarks.html
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
$8 = Desktop
$9 = dirlist-bin.txt
```

En este ejemplo, el comodín * se expande en 80 argumentos. ¿Como podemos procesar tal cantidad?
El shell ofrece un método, aunque de mala gana, para hacerlo. El comando `shift` hace que todos los parámetros "bajen uno" cada vez que se ejecute. De hecho, usando `shift`, es posible hacerlo con sólo un parámetro (además de `$0`, que nunca cambia):

```
#!/bin/bash
# posit-param2: script to display all arguments
count=1
while [[ $# -gt 0 ]]; do
echo "Argument $count = $1"
count=$((count + 1))
shift
done
```

Cada vez que se ejecuta `shift`, el valor de `$2` se mueve a `$1`, el valor de `$3` se mueve a `$2` y así sucesivamente. El valor de `$#` se reduce también en uno.

En el programa `posit-param2`, creamos un bucle que evaluaba el número de argumentos restantes y continuaba hasta que quedara al menos uno. Mostramos el argumento actual, incrementeamos la variable `count` con cada iteración del bucle para conseguir un contador del número de argumentos procesados y, finalmente, ejecutamos un `shift` para cargar `$1` con el siguiente argumento. Aquí está el programa en funcionamiento:

```
[me@linuxbox ~]$ posit-param2 a b c d
Argument 1 = a
Argument 2 = b
Argument 3 = c

Argument 4 = d
```

Aplicaciones simples

Incluso sin `shift`, es posible escribir aplicaciones útiles usando parámetros posicionales. A modo de ejemplo, aquí tenemos un programa de información de archivo simple:

```
#!/bin/bash
# file_info: simple file information program
PROGNAME=$(basename $0)
if [[ -e $1 ]]; then
echo -e "\nFile Type:"
file $1
echo -e "\nFile Status:"
stat $1
else
echo "$PROGNAME: usage: $PROGNAME file" >&2
exit 1
fi
```

Este programa muestra el tipo de archivo (determinado por el comando `file`) y el estado del archivo (del comando `stat`) de un archivo especificado. Una función interesante de este programa es la variable `PROGNAME`. Se le da el valor resultante del comando basename `$0`. El comando `basename` elimina la parte delantera de una ruta, dejando sólo el nombre base de un archivo. En nuestro ejemplo, `basename` elimina la parte de la ruta contenida en el parámetro `$0`, la ruta completa de nuestro programa de ejemplo. Este valor es útil cuando construimos mensajes tales como el mensaje de uso al final del programa. Escribiendo el código de esta forma, el script puede ser renombrado y el mensaje se ajusta automáticamente para contener el nombre del programa.

Usando parámetros posicionales con funciones de shell

Además de usar los parámetros posicionales para pasar argumentos a scripts de shell, pueden usarse para pasar argumentos a funciones de shell. Para comprobarlo, convertiremos el script `file_info` en una función de shell:

```
file_info () {
# file_info: function to display file information
if [[ -e $1 ]]; then
echo -e "\nFile Type:"
file $1
```

```
echo -e "\nFile Status:"
stat $1
else
echo "$FUNCNAME: usage: $FUNCNAME file" >&2
return 1
fi
}
```

Ahora, si un script que incorpora la función de shell `file_info` llama a la función con un argumento de nombre de archivo, el argumento pasará a la función.

Con esta capacidad, podemos escribir muchas funciones de shell útiles que no sólo pueden ser usadas con scripts, si no también dentro del archivo `.bashrc`.

Fíjate que la variable `PROGNAME` ha cambiado a la variable de shell `FUNCNAME`. El shell actualiza automáticamente esta variable para mantener el control de la función de shell ejecutada actualmente. Fíjate que `$0` siempre contiene la ruta completa del primer elemento de la linea de comandos (p.ej., el nombre del programa) y no contiene el nombre de la función de shell como podríamos esperar.

Manejando parámetros posicionales en masa

A veces es útil manejar todos los parámetros posicionales como un grupo. Por ejemplo, podríamos querer escribir un "envoltorio" alrededor de otro programa. Esto significa que creamos un script o función de shell que simplifique la ejecución de otro programa. El envoltorio proporciona una lista de opciones arcaicas de la línea de comandos y luego pasa una lista de argumentos al programa de menor nivel.

El shell ofrece dos parámetros especiales para este propósito. Ambos se expanden en la lista completa de parámetros posicionales, pero difieren en cosas muy sutiles. Son:

*Tabla 32-1: Los parámetros especiales * y @*

Parámetro	Descripción
`$*`	Se expande en la lista de parámetros posicionales, comenzando por 1. Cuando lo incluimos en comillas dobles, se expande en una cadena con comillas dobles conteniendo todos los parámetros posicionales, cada uno separado por el primer carácter de la variable de shell IFS (por defecto un carácter espacio)
`$@`	Se expande en la lista de parámetros posicionales, comenzando por 1. Cuando lo incluimos en comillas dobles, expande cada parámetro posicional en una palabra separada incluida entre comillas dobles.

Aquí tenemos un script que muestra estos parámetros especiales en acción:

```
#!/bin/bash
# posit-params3 : script to demonstrate $* and $@
print_params () {
echo "\$1 = $1"
echo "\$2 = $2"
echo "\$3 = $3"
```

```
echo "\$4 = $4"
}
pass_params () {
echo -e "\n" '$* :'; print_params $*
echo -e "\n" '"$*" :'; print_params "$*"
echo -e "\n" '$@ :'; print_params $@
echo -e "\n" '"$@" :'; print_params "$@"
}
pass_params "word" "words with spaces"
```

En este programa tan complicado, creamos dos argumentos: "word" y "words con spaces", y los pasamos a la función `pass_params`. Esa función, por turnos, los pasa a la función `print_params`, usando cada uno de los cuatro métodos disponibles con los parámetros especiales `$!` y `$@`. Cuando lo ejecutamos, el script revela las diferencias:

```
[me@linuxbox ~]$ posit-param3
$* :
$1 = word
$2 = words
$3 = with
$4 = spaces
"$*" :
$1 = word words with spaces
$2 =
$3 =
$4 =
$@ :
$1 = word
$2 = words
$3 = with
$4 = spaces
"$@" :
$1 = word
$2 = words with spaces
$3 =
$4 =
```

Con nuestros argumentos, tanto `$!` como `$@` producen un resultado de cuatro palabras:

```
word words with spaces
"$*" produces a one word result:
"word words with spaces"
"$@" produces a two word result:
"word" "words with spaces"
```

que coincide con nuestra intención real. La lección que aprendemos de esto es que aunque el shell proporciona cuatro formas diferentes de obtener la lista de parámetros posicionales, `"$@"` es de lejos la más útil para la mayoría de los casos, porque conserva la integridad de cada parámetro posicional.

Una aplicación más completa

Tras un largo paréntesis vamos a retomar nuestro trabajo con nuestro programa `sys_info_page`. Nuestra próxima mejora añadirá varias opciones de línea de comandos al programa tal como sigue:

- **Salida de archivo.** Añadiremos una opción para especificar un nombre para un archivo que contenga la salida del programa. Se especificará como `-f archivo` o como `--file archivo`.
- **Modo interactivo.** Esta opción preguntará al usuario un nombre para el archivo de salida y comprobará si el archivo especificado ya existe. Si es así, el usuario será preguntado antes de que el archivo existente se sobrescriba. Esta opción se especificara como `-i` o como `--interactive`.
- **Ayuda.** Tanto `-h` como `--help` debe ser especificada para hacer el programa para producir un mensaje de uso interactivo.

Aquí está el código necesario para implementar el procesado en la línea de comandos:

```
usage () {
echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
return
}
# process command line options
interactive=
filename=
while [[ -n $1 ]]; do
case $1 in
-f | --file) shift
filename=$1
;;
-i | --interactive) interactive=1
;;
-h | --help) usage
exit
;;
*) usage >&2
exit 1
;;
esac
shift
done
```

Primero, añadimos una función de shell llamada `usage` para mostrar un mensaje cuando se invoca la opción de ayuda o se intenta una opción desconocida.

A continuación, comenzamos el bucle de procesamiento. Este bucle continua mientras que el parámetro posicional `$1` no esté vacío. Al final del bucle, tenemos un comando `shift` para avanzar el parámetro posicional y asegurarnos de que el bucle finalmente terminará.

Dentro del bucle, tenemos una sentencia `case` que examina el parámetro posicional actual para ver si coincide con alguna de las opciones soportadas. Si encuentra un parámetro soportado, se actúa sobre él. Si no, se muestra el mensaje de uso y el script termina con un error.

El parámetro `-f` se maneja de una forma interesante. Cuando se detecta, hace que ocurra un `shift` adicional, que avanza el parámetro posicional `$1` al argumento de nombre de archivo proporcionado por la opción `-f`.

A continuación añadimos el código para implementar el modo interactivo:

```
# interactive mode
if [[ -n $interactive ]]; then
while true; do
read -p "Enter name of output file: " filename
if [[ -e $filename ]]; then
read -p "'$filename' exists. Overwrite? [y/n/q] > "
case $REPLY in
Y|y) break
;;
Q|q) echo "Program terminated."
exit
;;
*) continue
;;
esac
elif [[ -z $filename ]]; then
continue
else
break
fi
done
fi
```

La variable `interactive` no está vacía, y comienza un bucle infinito, que contiene el prompt del nombre del archivo y el consiguiente código existente de manejo de archivos. Si el archivo de salida ya existe, se pregunta al usuario si quiere sobrescribir, elegir otro nombre de archivo o salir del programa. Si el usuario elige sobrescribir un archivo existente, se ejecuta un `break` para terminar el bucle. Fíjate cómo la sentencia `case` sólo detecta si el usuario sobrescribe o sale. Cualquier otra opción hace que el bucle continué y pregunte al usuario de nuevo.

Para implementar la función de salida de nombre de archivo, deberíamos convertir primero el código existente de escritura de página en un función de shell, por las razones que veremos claras en un momento:

```
write_html_page () {
cat <<- _EOF_
<HTML>
<HEAD>
<TITLE>$TITLE</TITLE>
</HEAD>
<BODY>
<H1>$TITLE</H1>
<P>$TIMESTAMP</P>
$(report_uptime)
$(report_disk_space)
$(report_home_space)
</BODY>
</HTML>
_EOF_
return
```

```
}
# output html page
if [[ -n $filename ]]; then
if touch $filename && [[ -f $filename ]]; then
write_html_page > $filename
else
echo "$PROGNAME: Cannot write file '$filename'" >&2
exit 1
fi
else
write_html_page
fi
```

El código que maneja la lógica de la opción `-f` aparece al final de la lista mostrada anteriormente. En él, comprobamos la existencia de un nombre de archivo, y si encuentra uno, se realiza un test para ver si el archivo es modificable. Para hacer esto, se realiza un `touch`, seguido de un test para determinar si el archivo resultante es un archivo normal. Estos dos tests se encargan de situaciones donde se le indica una ruta no válida (`touch` fallará), y si el archivo ya existe, que sea un archivo normal.

Como podemos ver, se llama a la función `write_html_page` para realizar la generación de la página. Su salida es dirigida a la salida estándar (si la variable `file-name` está vacía) o redirigida a un archivo especificado.

Resumiendo

Con la inclusión de los parámetros posicionales, ahora podemos escribir scripts bastante funcionales. Para tareas simples y repetitivas, los parámetros posicionales hacen posible escribir funciones de shell muy útiles que pueden guardarse en el archivo `.bashrc`.

Nuestro programa `sys_info_page` ha crecido en complejidad y sofisticación. Aquí tenemos un listado completo , con los cambios más recientes resaltados:

```
#!/bin/bash
# sys_info_page: program to output a system information page
PROGNAME=$(basename $0)
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated $CURRENT_TIME, by $USER"
report_uptime () {
cat <<- _EOF_
<H2>System Uptime</H2>
<PRE>$(uptime)</PRE>
_EOF_
return
}
report_disk_space () {
cat <<- _EOF_
<H2>Disk Space Utilization</H2>
<PRE>$(df -h)</PRE>
_EOF_
```

```
return
}
report_home_space () {
if [[ $(id -u) -eq 0 ]]; then
cat <<- _EOF_
<H2>Home Space Utilization (All Users)</H2>
<PRE>$(du -sh /home/*)</PRE>
_EOF_
else
cat <<- _EOF_
<H2>Home Space Utilization ($USER)</H2>
<PRE>$(du -sh $HOME)</PRE>
_EOF_
fi
return
}
usage () {
echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
return
}
write_html_page () {
cat <<- _EOF_
<HTML>
<HEAD>
<TITLE>$TITLE</TITLE>
</HEAD>
<BODY>
<H1>$TITLE</H1>
<P>$TIMESTAMP</P>
$(report_uptime)
$(report_disk_space)
$(report_home_space)
</BODY>
</HTML>
_EOF_
return
}
# process command line options
interactive=
filename=
while [[ -n $1 ]]; do
case $1 in
-f | --file) shift
filename=$1
;;
-i | --interactive) interactive=1
;;
-h | --help) usage
exit
;;
*) usage >&2
exit 1
;;
```

```
esac
shift
done
# interactive mode
if [[ -n $interactive ]]; then
while true; do
read -p "Enter name of output file: " filename
if [[ -e $filename ]]; then
read -p "'$filename' exists. Overwrite? [y/n/q] > "
case $REPLY in
Y|y) break
;;
Q|q) echo "Program terminated."
exit
;;
*) continue
;;
esac
fi
done
fi
# output html page
if [[ -n $filename ]]; then
if touch $filename && [[ -f $filename ]]; then
write_html_page > $filename
else
echo "$PROGNAME: Cannot write file '$filename'" >&2
exit 1
fi
else
write_html_page
fi
```

Aún no hemos terminado. Todavía hay más cosas que podemos hacer y mejoras que podemos realizar.

Para saber más

- La *Bash Hackers Wiki* tiene un buen artículo sobre parámetros posicionales: http://wiki.bash-hackers.org/scripting/posparams
- El *Manual de Referencia de Bash* tiene un artículo sobre parámetros especiales, incluyendo `$*` y $@: http://www.gnu.org/software/bash/manual/bashref.html#Special-Parameters
- Además de las técnicas vistas en este capítulo, `bash` incluye un comando interno llamado `getopts`, que también puede usarse para procesar argumentos de línea de comandos. Está descrito en la sección SHELL BUILTIN COMMAND de la man page de `bash` y en la *Bash Hackers Wiki*: http://wiki.bash-hackers.org/howto/getopts_tutorial

Control de flujo: Bucles con for

En este último capítulo sobre control de flujo, veremos otra de las estructuras de bucles del shell. El *bucle for* difiere de los bucles while y until en que ofrece un medio de procesado de secuencias durante un bucle. Esto se vuelve muy útil cuando estamos programando. Por consiguiente, el bucle for es una estructura muy popular en scripting de `bash`.

Un bucle for se implementa, básicamente , con el comando `for`. En versiones modernas de `bash`, `for` está disponible en dos formas.

for: Forma tradicional del shell

La sintaxis original del comando `for` es:

```
for variable [in palabras]; do
comandos
done
```

Donde *variable* es el nombre de la variable que se incrementará durante la ejecución del bucle, *palabras* es una lista opcional de elementos que se asignarán secuencialmente a *variable*, y comandos son los *comandos* que se van a ejecutar en cada iteración del bucle.

El comando `for` es útil en la línea de comandos. Podemos comprobar fácilmente cómo funciona:

```
[me@linuxbox ~]$ for i in A B C D; do echo $i; done
A
B
C
D
```

En este ejemplo, se le da a `for` una lista de cuatro palabras: "A", "B", "C" y "D". Con una lista de cuatro palabras, el bucle se ejecuta cuatro veces. Cada vez que se ejecuta el bucle, se asigna una palabra a la variable `i`, Dentro del bucle, tenemos un comando `echo` que muestra el valor de i para mostrar la asignación. Igual que los bucles `while` y `until`, la palabra done cierra el bucle.

La característica realmente potente de `for` es el número de formas interesantes de crear listas de palabras. Por ejemplo, a través de expansión con llaves:

```
[me@linuxbox ~]$ for i in {A..D}; do echo $i; done
A
B
C
D
```

or expansión de rutas:

```
[me@linuxbox ~]$ for i in distros*.txt; do echo $i; done
distros-by-date.txt
distros-dates.txt
distros-key-names.txt
distros-key-vernums.txt
distros-names.txt
distros.txt
distros-vernums.txt
```

```
distros-versions.txt
```

o sustitución de comandos:

```
#!/bin/bash
# longest-word : find longest string in a file
while [[ -n $1 ]]; do
if [[ -r $1 ]]; then
max_word=
max_len=0
for i in $(strings $1); do
len=$(echo $i | wc -c)
if (( len > max_len )); then
max_len=$len
max_word=$i
fi
done
echo "$1: '$max_word' ($max_len characters)"
fi
shift
done
```

En este ejemplo, buscamos la cadena más larga dentro de una archivo. Una vez que le damos uno o más nombres de archivo en la línea de comandos, este programa usa el programa `strings` (que está incluido en el paquete GNU binutils) para generar una lista de "words" en texto legible en cada archivo. El bucle `for` procesa cada palabra por turnos y determina si la palabra actual es la más larga encontrada hasta ahora. Cuando concluye el bucle, se muestra la palabra más larga.

Si la porción opcional `in` *`words`* del comando `for` se omite, `for` por defecto procesa los parámetros posicionales. Modificaremos nuestro script `longest-word` para usar este método:

```
#!/bin/bash
# longest-word2 : find longest string in a file
for i; do
if [[ -r $i ]]; then
max_word=
max_len=0
for j in $(strings $i); do
len=$(echo $j | wc -c)
if (( len > max_len )); then
max_len=$len
max_word=$j
fi
done
echo "$i: '$max_word' ($max_len characters)"
fi
done
```

Como podemos ver, hemos cambiado el loop más externo para usar `for` en lugar de `while`. Omitiendo la lista de palabras en el comando `for`, se usan los parámetros posicionales en su lugar. Dentro del bucle, las instancias previas de la variable `i` han sido cambiadas a la variable `j`. El uso

de `shift` también ha sido eliminado.

¿Por qué i?

Habrás notado que se ha elegido la variable `i` para cada uno de los ejemplos del bucle `for` anteriores. ¿Por qué? No hay una razón específica en realidad, sólo la tradición. La variable utilizada con `for` puede ser cualquier variable válida, pero `i` es la más común, seguida de `j` y `k`.

El origen de esta tradición viene del lenguaje de programación Fortran. En Fortran, las variables no declaradas que comiencen con las letras I, J, K, L y M son automáticamente tratadas como enteros, mientras que las variables que comiencen con cualquier otra letra se tratan com reales (números con fracciones decimales). Este comportamiento llevó a los programadores a usar las variables I, J y K para variables de bucles, ya que era menos trabajoso usarlas cuando se necesitaba una variable temporal (y las variables de bucles lo son a menudo).

También llevó al siguiente chiste basado en Fortran:

"DIOS es real, a menos que se le declare como entero".

for: Forma del lenguaje C

Versiones recientes de `bash` han añadido una segunda forma de la sintaxis del comando `for`, una que se parece a la forma que encontramos en el lenguaje de programación C. Muchos otros lenguajes soportan esta forma también:

```
for (( expresión1; expresión2; expresión3 )); do
comandos
done
```

donde *expresión1*, *expresión2* y *expresión3* son expresiones aritméticas y *comandos* son los comandos a ejecutar durante cada iteración del bucle.

En términos de comportamiento, esta forma es equivalente a la siguiente estructura:

```
(( expresión1 ))
while (( expresión2 )); do
comandos
(( expresión3 ))
done
```

expresión1 se usa para inicializar condiciones para el bucle, *expresión2* se usa para determinar cuando termina el bucle y *expresión3* se ejecuta al funal de cada iteracion del bucle.

Aquí tenemos una aplicación típica:

```
#!/bin/bash
# simple_counter : demo of C style for command
for (( i=0; i<5; i=i+1 )); do
echo $i
done
```

Cuando lo ejecutamos, produce la siguiente salida:

```
[me@linuxbox ~]$ simple_counter
0
1
2
3
4
```

En este ejemplo, *expresion1* inicicaliza la variable `i` con el valor cero, *expresión2* permite al bucle continuar mientras que el valor de `i` permanezca menor que 5, y *expresión3* incrementa el valor de `i` en uno cada vez que se repite el bucle.

La forma de lenguaje C de `for` es útil siempre que necesitemos una secuencia numérica. Veremos varias aplicaciones de esto en los siguientes dos capítulos.

Resumiendo

Con nuestro conocimiento del comando `for`, aplicaremos ahora las mejoras finales a nuestro script `sys_info_page`. Actualmente, la función `report_home_space` aparece así:

```
report_home_space () {
if [[ $(id -u) -eq 0 ]]; then
cat <<- _EOF_
<H2>Home Space Utilization (All Users)</H2>
<PRE>$(du -sh /home/*)</PRE>
_EOF_
else
cat <<- _EOF_
<H2>Home Space Utilization ($USER)</H2>
<PRE>$(du -sh $HOME)</PRE>
_EOF_
fi
return
}
```

A continuación, la reescribiremos para proporcionar más detalle para cada directorio home de usuario, e incluiremos el número total de archivos y subdirectorios en cada uno de ellos:

```
report_home_space () {
local format="%8s%10s%10s\n"
local i dir_list total_files total_dirs total_size user_name
if [[ $(id -u) -eq 0 ]]; then
dir_list=/home/*
user_name="All Users"
else
dir_list=$HOME
user_name=$USER
fi
echo "<H2>Home Space Utilization ($user_name)</H2>"
for i in $dir_list; do
total_files=$(find $i -type f | wc -l)
```

```
total_dirs=$(find $i -type d | wc -l)
total_size=$(du -sh $i | cut -f 1)
echo "<H3>$i</H3>"
echo "<PRE>"
printf "$format" "Dirs" "Files" "Size"
printf "$format" "----" "-----" "----"
printf "$format" $total_dirs $total_files $total_size
echo "</PRE>"
done
return
}
```

Esta reescritura aplica mucho de lo que hemos aprendido hasta ahora. Aún probamos el superusuario, pero en lugar de realizar la lista completa de acciones como parte del `if`, configuramos algunas variables usadas posteriormente en un bucle `for`. Hemos añadido varias variables locales a la función y hecho uso de `printf` para formatear parte de la entrada.

Para saber más

- La *Guía Avanzada de Scripting en Bash* tiene un capítulo sobre bucles, con variedad de ejemplos usado for:
  http://tldp.org/LDP/abs/html/loops1.html
- El *Manual de Referencia de Bash* describe los comandos compuestos para bucles, incluyendo `for`:
  http://www.gnu.org/software/bash/manual/bashref.html#Looping-Constructs

Cadenas y números

Todos los programas de ordenador trabajan con datos. En los capítulos anteriores, nos hemos enfocado en procesamiento de datos a nivel de archivo. Sin embargo, muchos problemas de programación necesitan solventarse usando unidades de datos más pequeñas como cadenas y números.

En este capítulo, veremos varias funcionalidades del shell que se usan para manejar cadenas y números. El shell proporciona una variedad de expansiones de parámetros que realizan operaciones con cadenas. Además de la expansión aritmética (que vimos en el Capítulo 7), hay un programa de línea de comandos muy común llamado `bc`, que realiza matemáticas de alto nivel.

Expansión de parámetros

Aunque la expansión de parámetros surgió en el Capítulo 7, no lo vimos en detalle porque la mayoría de las expansiones de parámetros se usan en scripts en lugar de en la línea de comandos. Ya hemos trabajado con algunas formas de expansión de parámetros; por ejemplo, las variables de shell. El shell ofrece muchas más.

Parámetros básicos

La forma más simple de expansión de parámetros se refleja en el uso ordinario de variables. Por ejemplo:

```
$a
```

cuando se expande, se convierte en lo que contenga la variable. Los parámetros simples también pueden incluirse entre llaves:

```
${a}
```

Esto no tiene efecto en la expansión, pero se requiere si la variable es adyacente a otro texto, que pueda confundir al shell. En este ejemplo, intentaremos crear un nombre de archivo añadiendo la cadena "_file" al contenido de la variable `a`.

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

Si ejecutamos esta secuencia, el resultado será nada, porque el shell intentará expandir la variable `a_file` en lugar de `a`. Este problema puede solucionarse añadiendo llaves:

```
[me@linuxbox ~]$ echo "${a}_file"
foo_file
```

También hemos visto que podemos acceder a los parámetros posicionales mayores de 9 incluyendo el número entre llaves. Por ejemplo, para acceder al parámetro posicional undécimo, podemos hacer esto:

```
${11}
```

Expansiones para manejar variables vacías

Varias expansiones de parámetros manejan variables inexistentes o vacías. Estas expansiones son útiles para manejar parámetros posicionales perdidos y asignar valores por defecto a parámetros.

`${`*`parámetro`*`:-`*`palabra`*`}`

Si *parámetro* está sin definir (p.ej., no existe) o está vacío, esta expansión tiene como resultado el valor de *palabra*. Si *parámetro* no está vacío, la expansión tiene como resultado el valor de *parámetro*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
substitute value if unset
[me@linuxbox ~]$ echo $foo
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

`${`*`parámetro`*`:=`*`palabra`*`}`

Si *parámetro* está sin definir o vacío, esta expansión tiene como resultado el valor de *palabra*. Además, el valor de *palabra* se asigna a *parámetro*. Si *parámetro* no está vacío, la expansión tiene como resultado el valor de *parámetro*.

```
[me@linuxbox ~]$ foo=
```

```
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
default value if unset
[me@linuxbox ~]$ echo $foo
default value if unset
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

Nota: Los posicionales y otros parámetros no pueden asignarse de esta forma.

${*parámetro*:?*palabra*}

Si *parámetro* no está establecido o está vacío, esta expansión hace que el script termine con un error, y el contenido de *palabra* se envíe al error estándar. Si *parámetro* no está vacío, la expansión da como resultado el valor de *parámetro*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bar
[me@linuxbox ~]$ echo $?
0
```

${*parámetro*:+*palabra*}

Si *parámetro* no está establecido o está vacío, la expansión tiene como resultado nada. Si *parámetro* no está vacío, el valor de *palabra* se sustituye por *parámetro*; sin embargo, el valor de *parámetro* no cambia.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}

substitute value if set
```

Expansiones que devuelven nombres de variables

El shell tiene la capacidad de devolver los nombres de las variables. Esto se usa en algunas situaciones algo exóticas.

${!*prefijo**}
${!*prefijo*@}

Esta expansión devuelve los nombres de variables existentes con nombres que empiecen con

prefijo. Según la documentación de `bash`, ambas formas de expansión se comportan idénticamente. Aquí, listamos ttodas las variables en el entorno con nombres que comiencen con BASH:

```
[me@linuxbox ~]$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION
BASH_COMPLETION_DIR BASH_LINENO BASH_SOURCE BASH_SUBSHELL

BASH_VERSINFO BASH_VERSION
```

Operaciones con cadenas

Hay una gran colección de expansiones que pueden usarse para operar con cadenas. Muchas de estas expansiones son particularmente adecuadas para operaciones con rutas.

${#*parámetro*}

se expande en la longitud de la cadena contenida en *parámetro*. Normalmente, *parámetro* es una cadena; sin embargo, si *parámetro* es @ o *, la expansión da como resultado el número de parámetros posicionales.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo "'$foo' is ${#foo} characters long."
'This string is long.' is 20 characters long.
```

${*parámetro*:*margen*}
${*parámetro*:*margen*:*longitud*}

Estas expansiones se usan para extraer una porción de la cadena contenida en *parámetro*. La extracción comienza en *margen* caracteres desde el principio de la cadena y continua hasta el final de la cadena, a no ser que se especifique *longitud*.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo:5}
string is long.
[me@linuxbox ~]$ echo ${foo:5:6}
string
```

Si el valor de *margen* es negativo, se considera que empieza por el final de la cadena en lugar de por el principio. Fíjate que los valores negativos deben ir precedidos por un espacio para evitar la confusión con la expansión `${parámetro:-palabra}`. *longitud*, si está presente, no debe ser menor de cero.

Si *parámetro* es @, el resultado de la expansión es *longitud* parámetros posicionales, comenzando en *margen*.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo: -5}
long.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

${*parámetro#patrón*}
${*parámetro##patrón*}

Estas expansiones eliminan la parte delantera de la cadena contenida en *parámetro* definida por *patrón*. *patrón* es un patrón comodín como los que se usan en expansiones de rutas. La diferencia entre las dos formas es que la # elimina el resultado más corto, mientras que la forma ## elimina el resultado más largo.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo#*.}
txt.zip
[me@linuxbox ~]$ echo ${foo##*.}
zip
```

${*parámetro%patrón*}
${*parámetro%%patrón*}

Estas expansiones son iguales que las # y ## anteriores, excepto que eliminan texto desde el final de la cadena contenida en *parámetro* en lugar que desde el principio.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo%.*}
file.txt
[me@linuxbox ~]$ echo ${foo%%.*}
file
```

${*parámetro/patrón/cadena*}
${*parámetro//patrón/cadena*}
${*parámetro/#patrón/cadena*}
${*parámetro/%patrón/cadena*}

Esta expansión realiza un "buscar y reemplazar" dentro del contenido de *parámetro*. Si se encuentra texto que coincida con el comodín *patrón*, se reemplaza con el contenido de *cadena*. En la forma normal, solo se reemplaza la primera coincidencia de patrón. En la forma //, se reemplazan todas las coincidencias. La forma /# requiere que la coincidencia ocurra al principio de la cadena, y la forma /% requiere que la coincidencia ocurra al final de la cadena. /*cadena* puede omitirse, lo que causa que el texto señalado por patrón se borre.

```
[me@linuxbox ~]$ foo=JPG.JPG
[me@linuxbox ~]$ echo ${foo/JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo//JPG/jpg}
jpg.jpg
[me@linuxbox ~]$ echo ${foo/#JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo/%JPG/jpg}
JPG.jpg
```

La expansión de parámetros es algo bueno de conocer. Las expansiones con manipulación de cadenas pueden usarse como sustitutos de otros comandos comunes como `sed` o `cut`. Las expansiones mejoran la eficiencia de los scripts eliminando el uso de programas externos. Como ejemplo, modificaremos el programa `longest-word` que vimos en el capítulo anterior para usar la expansión de parámetros `${#j}` en lugar de la sustitución de comandos `$(echo $j | wc -c)` y su subshell resultante, así:

```
#!/bin/bash
# longest-word3 : find longest string in a file
for i; do
if [[ -r $i ]]; then
max_word=
max_len=
for j in $(strings $i); do
len=${#j}
if (( len > max_len )); then
max_len=$len
max_word=$j
fi
done
echo "$i: '$max_word' ($max_len characters)"
fi
shift
done
```

A continuación, compararemos la eficiencia de las dos versiones usando el comando `time`:

```
[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)
real 0m3.618s
user 0m1.544s
sys 0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)
real 0m0.060s
user 0m0.056s
sys 0m0.008s
```

La versión original del script tarda 3,618 segundos en escanear el archivo, mientras que la nueva versión, usando expansión de parámetros, tarda sólo 0,06 segundos - una mejora significativa.

Conversión de mayúsculas y minúsculas

Las versiones recientes de bash tiene soporte para conversión mayúsculas/minúsculas de cadenas. `bash` tiene cuatro expansiones de parámetros y dos opciones para el comando `declare` para soportarlo.

¿Y para que sirve la conversión mayúsculas-minúsculas? Además del obvio valor estético, tiene un

importante papel en programación. Consideremos el caso de una búsqueda en base de datos. Imagina que un usuario ha introducido una cadena en un campo de entrada de texto que querernos buscar en una base de datos. Es posible que el usuario introduzca el valor todo en mayúsculas o todo en minúsculas o una combinación de ambas. Ciertamente no queremos abarrotar nuestra base de datos con cada combinación posible de letras mayúsculas y minúsculas. ¿qué hacemos?

Una aproximación común a este problema es *normalizar* la entrada del usuario. O sea, convertirla en una forma estandarizada antes de intentar la búsqueda en la base de datos. Podemos hacer esto convirtiendo todos los caracteres de la entrada del usuario a mayúsculas o minúsculas y asegurarnos de que las entradas en la base de datos están normalizadas de la misma forma.

El comando `declare` puede usarse para normalizar cadenas a mayúsculas o minúsculas. Usando `declare`, podemos forzar que una variable siempre contenga el formato deseado sin importar lo que tenga asignado:

```
#!/bin/bash
# ul-declare: demonstrate case conversion via declare
declare -u upper
declare -l lower
if [[ $1 ]]; then
upper="$1"
lower="$1"
echo $upper
echo $lower
fi
```

En el script anterior, usamos `declare` para crear dos variables, `upper` y `lower`. Asignamos el valor del primer argumento de la línea de comandos (parámetro posicional 1) a cada variable y luego mostrarlas en la pantalla:

```
[me@linuxbox ~]$ ul-declare aBc
ABC
abc
```

Como podemos ver, el argumento de línea de comandos ("aBc") ha sido normalizado.

Hay cuatro expansiones de parámetros que realizan la conversión mayúsuclas/minúsculas:

Tabla 34-1: Expansión de parámetros para conversión mayúsculas/minúsculas

Formato	Resultado
${*parámetro*,,}	Expande el valor de *parámetro* todo en minúsculas.
${*parámetro*,}	Expande el valor de *parámetro* cambiando sólo el primer carácter a minúsculas.
${*parámetro*^^}	Expande el valor de *parámetro* todo en mayúsculas.
${*parámetro*^}	Expande el valor de *parámetro* cambiando sólo el primer carácter a mayúsculas (Nombre propio).

Aquí tenemos un script que demuestra estas expansiones:

```
#!/bin/bash
# ul-param - demonstrate case conversion via parameter expansion
if [[ $1 ]]; then
echo ${1,,}
echo ${1,}
echo ${1^^}
echo ${1^}
fi
```

Aquí está el script en acción:

```
[me@linuxbox ~]$ ul-param aBc
abc
aBc
ABC
ABc
```

De nuevo, procesamos el primer argumento de la línea de comandos y obtenemos las cuatro variantes soportadas por las expansiones de parámetros. Aunque este script usa el primer parámetro posicional, *parámetro* puede ser una cadena, una variable o una expresión.

Evaluación aritmética y expansión

Hemos visto la expansión aritmética en el Capítulo 7. Se usa para realizar varias operaciones aritméticas con enteros. Su forma básica es:

`$((expresión))`

donde *expresión* es una expresión aritmética válida.

Esto está relacionado con el comando compuesto `(( ))` usado para evaluación aritmética (pruebas de verdad) que vimos en el Capítulo 27.

En capítulos anteriores, vimos algunos de los tipos comunes de expresiones y operadores. Aquí, veremos una lista más completa.

Bases numéricas

Cuando estábamos en el Capítulo 9, echamos un vistazo a los octales (base 8) y hexadecimales (base 16). En las expresiones aritméticas, el shell soporta constantes enteras en cualquier base.

Tabla 34-2: Especificando diferentes bases numéricas

Notación	Descripción
número	Por defecto, los números sin notación se tratan como enteros decimales (base 10).
0número	En expresiones aritméticas,

	números que comienzan con un cero se consideran octales.
0xnúmero	Notación hexadecimal
base#número	*número* en base *base*

Algunos ejemplos:

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

En los ejemplos anteriores, mostramos el valor de un número hexadecimal `ff` (el mayor número de dos dígitos) y el mayor número binario (base 2) de ocho dígitos.

Operadores unarios

Hay dos operadores unarios, el `+` y el `-`, que se usan para indicar si un número es positivo o negativo, respectivamente. Por ejemplo, `-5`.

Aritmética simple

Los operadores aritméticos ordinarios se listan en la siguiente tabla:

Tabla 34-3: Operadores aritméticos

Operador	Descripción
+	Adición
-	Sustracción
*	Multiplicación
/	División entera
**	Exponenciación
%	Módulo (resto)

La mayoría son autoexplicativos, pero la división entera y el módulo requieren una explicación.

Como la aritmética del shell sólo opera con enteros, los resultados de la división son siempre números enteros:

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))
2
```

Esto hace que la determinación del resto de una división sea más importante:

```
[me@linuxbox ~]$ echo $(( 5 % 2 ))
1
```

Usando los operadores división y módulo, podemos determinar que 5 dividido entre 2 da como resultado 2, con un resto de 1.

Calcular el resto es útil en bucles. Permite que una operación se realice en intervalos especificados durante la ejecución del bucle. En el ejemplo siguiente, mostramos una línea de números, destacando cada múltiplo de 5:

```
#!/bin/bash
# modulo : demonstrate the modulo operator
for ((i = 0; i <= 20; i = i + 1)); do
remainder=$((i % 5))
if (( remainder == 0 )); then
printf "<%d> " $i
else
printf "%d " $i
fi
done
printf "\n"
```

Cuando se ejecuta, el resultado es el siguiente:

```
[me@linuxbox ~]$ modulo

<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

Asignación

A pesar de que su utilidad puede no ser inmediatamente aparente, las expresiones aritméticas pueden realizar asignación. Hemos realizado asignación muchas veces, pero en un contexto diferente. Cada vez que damos un valor a una variable, estamos realizando asignación. Podemos hacerlo también dentro de expresiones aritméticas:

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo $foo
[me@linuxbox ~]$ if (( foo = 5 ));then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ echo $foo
5
```

En el ejemplo anterior, primero asignamos un valor vacío a la variable `foo` y verificamos que está realmente vacía. A continuación, realizamos un if con el comando compuesto `(( foo = 5 ))`. Este proceso hace dos cosas interesantes: 1) asigna el valor 5 a la variable `foo`, y 2) se evalúa como verdadera porque se le asignó a `foo` un valor distinto de cero.

Nota: Es importante recordar el significado exacto del = en la expresión anterior. Un = individual realiza asignación. `foo = 5` dice "haz a `foo` igual a 5", mientras == evalúa equivalencia. `foo == 5` dice "¿es `foo` igual a 5? Esto puede ser muy confuso porque el comando `test` acepta un =

individual para equivalencia de cadenas. Esta es otra razón más para usa los comandos compuestos más modernos `[[ ]]` y `(( ))` en lugar de `test`.

Además del `=`, el shell también ofrece notaciones que realizan algunos asignaciones muy útiles:

Tabla 34-4: Operadores de asignación

Notación	Descripción
parámetro = valor	Asignación simple. Asigna *valor* a *parámetro*,
parámetro += valor	Adición. Equivale a *parámetro* = *parámetro* + *valor*.
parámetro -= valor	Sustracción. Equivale a *parámetro* = *parámetro* - *valor*.
*parámetro *= valor*	Multiplicación. Equivale a *parámetro* = *parámetro* * *valor*.
parámetro /= valor	División entera. Equivale a *parámetro* = *parámetro* / *valor*.
parámetro %= valor	Módulo. Equivale a *parámetro* = *parámetro* % *valor*.
parámetro ++	Variable post-incremental. Equivale a *parámetro* = *parámetro* +1 (de todas formas, lo veremos continuación).
parámetro --	Variable post-decremental. Equivale a *parámetro* = *parámetro* - 1.
++parámetro	Variable pre-incremental. Equivale a *parámetro* = *parámetro* +1.
--parámetro	Variable pre-decremental. Equivale a *parámetro* = *parámetro* -1.

Estos operadores de asignación aportan un atajo para muchas de las tareas aritméticas comunes. De especial interés son los operadores incrementales (`++`) y decrementales (`--`), que aumentan o disminuyen el valor de sus parámetros en uno. Este estilo de notación se toma del lenguaje de programación C y ha sido incorporado a otros lenguajes de programación, incluido `bash`.

Los operadores pueden aparecer delante o detrás de un parámetro. Aunque ambos aumentan o disminuyen el parámetro en uno, las dos localizaciones tiene una diferencia sutil. Si se coloca delante del parámetro , el parámetro aumenta (o disminuye) antes de que se devuelva el parámetro. Si se coloca después, la operación se realiza *después* de que el parámetro se devuelva. Esto es un poco extraño, pero está hecho a propósito. Aquí tenemos una demostración:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++))
1
[me@linuxbox ~]$ echo $foo
2
```

Si asignamos el valor de uno a la variable `foo` y luego la incrementamos con el operador `++` situado tras el nombre del parámetro, `foo` es devuelto con el valor de uno. Sin embargo, si miramos el valor de la variable una segunda vez, vemos el valor incrementado. Si colocamos el operador `++` delante del parámetro, tenemos el comportamiento esperado:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo))
```

```
2
[me@linuxbox ~]$ echo $foo
2
```

Para la mayoría de las aplicaciones de shell, colocar un prefijo al operador será lo más útil.

Los operadores `++` y `--` se usan a menudo junto con los bucles. Haremos algunas mejoras en nuestro módulo de script ajustándolo un poco:

```
#!/bin/bash
# modulo2 : demonstrate the modulo operator
for ((i = 0; i <= 20; ++i )); do
if (((i % 5) == 0 )); then
printf "<%d> " $i
else
printf "%d " $i
fi
done

printf "\n"
```

Operaciones con bits

Hay una clase de operadores que manipulan números de un modo inusual. Estos operadores funcionan a nivel de bits. Se usan para ciertos tipos de tareas de bajo nivel, a menudo implican configuración o lectura de banderas-bit:

Tabla 34-5: Operadores de bits

<table>
<tr><th>Operador</th><th>Descripción</th></tr>
<tr><td>~</td><td>Negación bit a bit. Niega todos los bit de un número.</td></tr>
<tr><td><<</td><td>Cambia bit a bit hacia la izquierda. Cambia todos los bits en un número hacia la izquierda.</td></tr>
<tr><td>>></td><td>Cambia bit a bit hacia la derecha. Cambia todos los bits en un número hacia la derecha.</td></tr>
<tr><td>&</td><td>AND bit a bit. Realiza una operación AND en todos los bits en dos números.</td></tr>
<tr><td>|</td><td>OR bit a bit. Realiza una operación OR en todos los bits en dos números.</td></tr>
<tr><td>^</td><td>XOR bit a bit. Realiza una operación OR exclusiva en todos los bits en dos números.</td></tr>
</table>

Fíjate que también hay los correspondientes operadores de asignación (por ejemplo, `<<==`) para todos excepto para la negación bit a bit.

Aquí lo comprobaremos produciendo una lista de potencias de 2, usando el operador de cambio bit a bit hacia la izquierda:

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done
12
48
```

```
16
32
64

128
```

Lógica

Como descubrimos en el Capítulo 27, el comando compuesto `(( ))` soporta una variedad de operadores de comparación. Hay unos pocos más que pueden usarse par evaluar lógica. Aquí tenemos la lista completa:

Tabla 34-6: Operadores de comparación

Operador	Descripción
`<=`	Menor o igual que
`>=`	Mayor o igual que
`<`	Menor que
`>`	Mayor que
`==`	Igual a
`!=`	No igual a
`&&`	AND lógico
`||`	OR lógico
`expr1``?`*`expr2`*`?:`*`expr3`*	Operación de comparación (ternario). Si *expr1* se evalúa como no-cero (verdad aritmética) entonces *expr2*, si no *expr3*.

Cuando se usan para operaciones lógicas, las expresiones siguen las reglas de la lógica aritmética; o sea, las expresiones que se evalúan como cero se consideran falsas, mientras que las expresiones no-cero se consideran verdaderas. El comando compuesto `(( ))` mapea los resultados dentro de los códigos de salida normales de shell:

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

El operador lógico más extraño es el *operador ternario*. Este operador (que se modeló después del que hay en el lenguaje de programación C) realiza un test lógico autónomo. Puede usarse como un tipo de sentencia if/then/else. Actúa en tres expresiones aritméticas (las cadenas no funcionarán), y si la primera expresión es verdadera (o no-cero) se ejecuta la segunda expresión. Si no, se ejecuta la tercera expresión. Podemos probarlo en la línea de comandos:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
1
```

```
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

Aquí vemos al operador ternario en acción. Este ejemplo implementa un interruptor. Cada vez que el operador se ejecuta, el valor de la variable a cambia de cero a uno o viceversa.

Por favor, fíjate que realizar asignación dentro de expresiones no es sencillo.

Cuando lo intentamos, bash devuelve un error:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: ((: a<1?a+=1:a-=1: attempted assignment to non-variable
(error
token is "-=1")
```

Este problema puede evitarse incluyendo la expresión de asignación entre paréntesis:

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

A continuación, vemos un ejemplo más completo del uso de operadores aritméticos en un script que produce una tabla simple de números:

```
#!/bin/bash
# arith-loop: script to demonstrate arithmetic operators
finished=0
a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"
until ((finished)); do
b=$((a**2))
c=$((a**3))
printf "%d\t%d\t%d\n" $a $b $c
((a<10?++a:(finished=1)))
done
```

En este script, implementamos un bucle until basado en el valor de la variable `finished`. Inicialmente, la variable está establecida en cero (falso aritmético) y continuamos el bucle hasta que se convierta en no-cero. Dentro del bucle, calculamos el cuadrado y el cubo de la variable contador `a`. Al final del bucle, el valor de la variable contador se evalúa. Si es menor de `10` (el número máximo de iteraciones), se incrementa en uno, si no, se le da a la variable `finished` el valor de uno, haciendo a `finished` aritméticamente verdadera, y de esta forma terminando el bucle. Ejecutar el script no da este resultado:

```
[me@linuxbox ~]$ arith-loop
a a**2 a**3
= ==== ====
0    0    0
1    1    1
2    4    8
3    9   27
```

```
4   16   64
5   25  125
6   36  216
7   49  343
8   64  512
9   81  729

10 100 1000
```

bc - Un lenguaje de cálculo de precisión arbitraria

Hemos visto como el shell puede manejar todos los tipos de aritmética con enteros, pero ¿qué pasa si necesitamos realizar matemáticas más complejas o incluso usar números de coma flotante? La respuesta es, no podemos. Al menos no directamente con el shell. Para hacerlo, necesitamos usar un programa externo. Hay varios enfoques que podemos tomar. Incluir programas Perl o AWK es una solución posible, pero desafortunadamente, están fuera del objetivo de este libro.

Otro acercamiento es usar un programa de cálculo especializado. Uno de estos programas que encontramos en la mayoría de los sistemas Linux se llama `bc`.

El programa bc lee un archivo escrito en su propio lenguaje tipo C y lo ejecuta. Un script `bc` puede ser una archivo por separado o puede leerse de la entrada estándar. El lenguaje `bc` tiene bastantes características incluyendo variables, bucles y funciones definidas por el programador. No veremos `bc` completamente aquí, sólo lo probaremos. `bc` está bien documentado por su man page.

Empecemos con un ejemplo simple. Escribiremos un script `bc` para sumar 2 más 2:

```
/* A very simple bc script */
2 + 2
```

La primera línea del script es un comentario. bc usa la misma sintáxis para comentarios que el lenguaje de programación C. Comentarios que se pueden expandir en múltiples líneas, comenzando con `/*` y finalizando con `/*`.

Usando bc

Si guardamos el script `bc` anterior como `foo.bc`, podemos ejecutarlo de la siguiente manera:

```
[me@linuxbox ~]$ bc foo.bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4
```

Si miramos con atención, podemos ver el resultado abajo del todo, tras el mensaje de copyright. Este mensaje puede suprimirse con la opción `-q` (quiet).

`bc` también puede usarse interactivamente:

```
[me@linuxbox ~]$ bc -q
```

```
2 + 2
4
quit
```

Cuando usamos `bc` interactivamente, simplemente tecleamos los cálculos que queremos realizar, y los resultados se muestran inmediatamente. El comando de `bc` `quit` finaliza la sesión interactiva.

También es posible pasar un script a `bc` a través de la entrada estándar:

```
[me@linuxbox ~]$ bc < foo.bc
4
```

La capacidad de tomar entrada estándar significa que podemos usar documentos-aquí, cadenas-aquí y entubados para pasar scripts. Aquí tenemos un ejemplo de cadena:

```
[me@linuxbox ~]$ bc <<< "2+2"

4
```

Un script de ejemplo

Como ejemplo de la vida real, construiremos un script que realice un cálculo común, el pago mensual de un préstamo. En el siguiente script, usamos un documento-aquí para pasar un script a `bc`:

```
#!/bin/bash
# loan-calc : script to calculate monthly loan payments
PROGNAME=$(basename $0)
usage () {
cat <<- EOF
Usage: $PROGNAME PRINCIPAL INTEREST MONTHS
Where:
PRINCIPAL is the amount of the loan.
INTEREST is the APR as a number (7% = 0.07).
MONTHS is the length of the loan's term.
EOF
}
if (($# != 3)); then
usage
exit 1
fi
principal=$1
interest=$2
months=$3
bc <<- EOF
scale = 10
i = $interest / 12
p = $principal
n = $months
a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
print a, "\n"
EOF
```

Cuando lo ejecutamos, el resultado aparece así:

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
1270.7222490000
```

Este ejemplo calcula el pago mensual de un préstamo de 135.000 $ al 7.75% TAE en 180 meses (15 años). Fíjate la precisión de la respuesta. Viene determinada por el valor dado a la variable especial `scale` en el script `bc`. La man page de `bc` proporciona una descripción completa del lenguaje de scripting de `bc`. Aunque su notación matemática es ligeramente diferente de la del shell (`bc` se parece más C), en su mayoría es muy familiar, basándonos en lo aprendido hasta ahora.

Resumiendo

En este capítulo, hemos aprendido muchas pequeñas cosas que pueden usarse para hacer "trabajo real" en scripts. A la vez que nuestra experiencia en scripts aumenta, la capacidad de manipular eficientemente cadenas y números nos será de mucho valor. Nuestro script `loan-calc` demuestra que incluso scripts simples pueden crearse para hacer cosas realmente útiles.

Crédito extra

Aunque la funcionalidad básica del script `loan-calc` está en su sitio, el script está lejos de estar completo. Como crédito extra, prueba a mejorar el script `loan-calc` con las siguientes características:

- Verificación completa de los argumentos de la línea de comandos
- Una opción de línea de comandos para implementar un modo "interactivo" que pregunte al usuario el principal, el tipo de interes y el plazo del prestamo.
- Un mejor formateo de la salida.

Para saber más

- La *Bash Hackers Wiki* tiene un buen tema sobre expansión de parámetros: http://wiki.bash-hackers.org/syntax/pe
- El *Manual de Referencia de Bash* lo trata también: http://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expansion
- *Wikipedia* tiene un buen artículo sobre las operaciones con bits: http://en.wikipedia.org/wiki/Bit_operation
- y un artículo sobre operaciones ternarias: http://en.wikipedia.org/wiki/Ternary_operation
- así como una descripción de la fórmula para calcular pagos de prestamos usada en nuestro script loan-calc:
- http://en.wikipedia.org/wiki/Amortization_calculator

Arrays

En el último capítulo, vimos como el shell puede manipular cadenas y números. El tipo de datos que hemos visto hasta ahora se conocen en los círculos informáticos como *variables escalares*; o sea, variables que contienen un valor individual.

En este capítulo, veremos otro tipo de estructura de datos llamados *arrays* (*vectores*), que contienen valores múltiples. Los arrays son una característica de prácticamente todos los lenguajes de

programación. El shell los soporta también, sólo que de una forma algo limitada. Incluso así, pueden ser muy útiles para resolver problemas de programación.

¿Qué son los arrays?

Los arrays son variables que almacenan más de un valor a la vez. Los arrays se organizan como una tabla. Consideremos una hoja de cálculo como ejemplo. Una hoja de cálculo funciona como un *array de dos dimensiones*. Tiene tanto filas como columnas, y una celda individual de la hoja de cálculo puede localizarse según su dirección de fila y columna. Un array se comporta de la misma forma. Un array tiene celdas, que se llaman *elementos*, y cada elemento contiene datos. Un elemento individual de un array es accesible usando una dirección llamada un *indice* o *subscript*.

La mayoría de lenguajes de programación soportan *arrays multidimensionales*. Una hoja de cálculo es un ejemplo de un array multidimensional de dos dimensiones, anchura y altura. Muchos lenguajes de programacion soportan arrays con un número arbitrario de dimensiones, aunque los arrays de dos y tres dimensiones son probablemente los más usados.

Los arrays en `bash` se limitan a una única dimension. Podemos pensar que son hojas de cálculo con una única columna. Incluso con esta limitación, hay muchas aplicaciones para ellos. El soporte para arrays apareció por primera vez en `bash` versión 2. El programa de shell original de unix, sh, no soporta arrays de ningún tipo.

Creando un array

Las variables array se nombran igual que otras variables de `bash`, y se crean automáticamente cuando se accede a ellas. Aquí tenemos un ejemplo:

```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

Aquí vemos un ejemplo tanto de asignación como de acceso a un elemento de un array. Con el primer comando, al elemento 1 del array a se le asigna el valor "foo". El segundo comando muestra el valor almacenado en el elemento 1. El uso de llaves en el segundo comando se requiere para evitar que el shell intente una expansión de ruta en el nombre del elemento del array.

Un array puede también crearse con el comando `declare`:

```
[me@linuxbox ~]$ declare -a a
```

Usando la opción `-a`, este ejemplo de `declare` crea el array `a`.

Asignando valores a un array

Los valores pueden asignarse de dos formas. Valores individuales pueden asignarse usando la siguiente sintaxis:

```
nombre[índice]=valor
```

donde *nombre* es el nombre del array e *índice* es un entero (o una expresión aritmética) mayor o

igual que cero. Fíjate que el primer elemento de un array es el índice cero, no uno. *valor* es una cadena o un entero asignado al elemento del array.

Valores múltiples pueden asignarse usando la siguiente sintaxis:

```
nombre=(valor1 valor2 ...)
```

donde *nombre* es el nombre del array y *valor...* son los valores asignados secuencialmente a elementos del array, comenzando por el elemento cero. Por ejemplo, si queremos asignar los días de las semana en abreviaturas al array days, podríamos hacer esto:

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

También es posible asignar valores a un elemento en concreto especificando un índice para cada valor:

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu

[5]=Fri [6]=Sat)
```

Accediendo a los elementos de un array

Entonces ¿para qué sirven los arrays? De la misma forma que muchas tareas de manejo de datos pueden realizarse con un programa de hojas de cálculo, muchas tareas de programación pueden realizarse con arrays.

Consideremos un ejemplo simple de recogida y presentación de datos. Construiremos un script que examine la hora de modificación de los archivos de un directorio determinado. Desde estos datos, nuestro script mostrará una tabla con la hora en que los datos fueron modificados por última vez. Dicho script podría usarse para determinar cuando está más activo un sistema. Este script, llamado `hours`, produce este resultado:

```
[me@linuxbox ~]$ hours .
Hour Files Hour Files
---- ----- ---- -----
 00     0   12    11
 01     1   13     7
 02     0   14     1
 03     0   15     7
 04     1   16     6
 05     1   17     5
 06     6   18     4
 07     3   19     4
 08     1   20     1
 09    14   21     0
 10     2   22     0
 11     5   23     0
Total files = 80
```

Ejecutamos el programa `hours`, especificando el directorio actual como destino. Produce una tabla mostrando, para cada hora del día (0-23), cuantos archivos han sido modificados por última vez. El código para producir esto es el que sigue:

```
#!/bin/bash
# hours : script to count files by modification time
usage () {
echo "usage: $(basename $0) directory" >&2
}
# Check that argument is a directory
if [[ ! -d $1 ]]; then
usage
exit 1
fi
# Initialize array
for i in {0..23}; do hours[i]=0; done
# Collect data
for i in $(stat -c %y "$1"/* | cut -c 12-13); do
j=${i/#0}
((++hours[j]))
((++count))
done
# Display data
echo -e "Hour\tFiles\tHour\tFiles"
echo -e "----\t-----\t----\t-----"
for i in {0..11}; do
j=$((i + 12))
printf "%02d\t%d\t%02d\t%d\n" $i ${hours[i]} $j ${hours[j]}
done
printf "\nTotal files = %d\n" $count
```

El script consiste en una función (`usage`) y un cuerpo principal con cuatro secciones. En la primera sección, comprobamos que hay un argumento en la línea de comandos y que es un directorio. Si no, mostramos el mensaje de uso y salimos.

La segunda sección inicializa el array `hours`. Lo hace asignando a cada elemento un valor cero. No hay ningún requerimiento especial para preparar arrays antes de usarlos, pero nuestro script necesita asegurarse de que ningún elemento se queda vacío. Fíjate la interesante forma en que el bucle se construye. Empleado expansión con llaves `({0..23})`, podemos generar fácilmente una secuencia de palabras para el comando `for`.

La siguiente sección recoge los datos ejecutando el programa `stat` en cada archivo del directorio. Usamos `cut` para extraer los dígitos de la hora del resultado. Dentro del bucle, necesitamos eliminar los ceros a la izquierda de nuestro campo hora, ya que el shell trata (y finalmente falla) de interpretar los valores del "00" al "99" como números octales (ver Tabla 34-1). A continuación, incrementamos el valor del elemento del array correspondiente a la hora del día. Finalmente, incrementamos un contador (`count`) para seguir la pista del número total de archivos en el directorio.

La última sección del script muestra el contenido del array. Primero mostramos un par de líneas de encabezado y luego entramos en un bucle que produce una salida en dos columnas. Finalmente, mostramos la lista completa de archivos.

Operaciones con arrays

Hay muchas operaciones comunes con arrays. Cosas como borrar arrays, determinar su tamaño, su orden, etc. tienen muchas aplicaciones en scripting.

Mostrando todo el contenido de un array

Los subscripts `*` y @ pueden usarse para acceder a todos los elementos de un array. Al igual que con los parámetros posicionales, la notación @ es la más útil de las dos. Aquí tenemos una prueba:

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in "${animals[*]}"; do echo $i; done
a dog a cat a fish
[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done
a dog
a cat
a fish
```

Creamos el array `animals` y le asignamos tres cadenas de dos palabras. Luego ejecutamos cuatro bucles para ver el efecto de la separación de palabras en el contenido del array. El comportamiento de las notaciones `${animals[*]}` y `${animals[@]}`es idéntico hasta que se entrecomillen. La notación `*` da como resultado una palabra individual con el contenido del array, mientras que la notación @ da como resultado tres palabras, lo que coincide con el contenido "real" del array.

Determinando el número de elementos de una array

Usando expansión de parámetros, podemos determinar el número de elementos en un array de forma muy parecida a determinar la longitud de una cadena. Aquí tenemos un ejemplo:

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]} # number of array elements
1
[me@linuxbox ~]$ echo ${#a[100]} # length of element 100
3
```

Creamos un array a y le asignamos la cadena "foo" al elemento 100. A continuación, usamos expansión de parámetros para examinar la longitud del array, usando la notación @. Finalmente,

vemos la longitud del elemento 100 que contiene la cadena "foo". Es interesante fijarse que mientras que asignamos nuestra cadena al elemento 100, bash sólo reporta un elemento en el array. Esto difiere del comportamiento de otros lenguajes en los que los elementos sin uso del array (elementos 0-99) serían inicializados con valores vacíos y se cuentan.

Encontrando los subscripts usados por un array

Como bash permite que los arrays contengan "huecos" en la asignación de subscripts, a veces es útil para determinar qué elementos existen en realidad. Esto puede hacerse con una expansión de parámetros usando las siguientes fórmulas:

```
${!array[*]}
${!array[@]}
```

donde *array* es el nombre de una variable array. Como en las otras expansiones que usan * y @, la forma @ entre comillas es la más útil, ya que se expande en palabras separadas:

```
[me@linuxbox ~]$ foo=([2]=a [4]=b [6]=c)
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo $i; done
ab
c
[me@linuxbox ~]$ for i in "${!foo[@]}"; do echo $i; done
24

6
```

Añadiendo elementos al final de un array

Saber el número de elementos de un array no ayuda si necesitamos añadir valores al final del array, ya que los valores devueltos por las notaciones * y @ no nos dicen el máximo índice del array en uso. Afortunadamente, el shell nos da una solución. Usando el operador de asignación +=, podemos añadir valores automáticamente al final de un array. Aquí, asignamos tres valores al array `foo`, y luego le añadimos tres más.

```
[me@linuxbox ~]$ foo=(a b c)
[me@linuxbox ~]$ echo ${foo[@]}
a b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]}

a b c d e f
```

Ordenando un array

Como en las hojas de cálculo, a menudo es necesario ordenar los valores de una columna de datos. El shell no tiene una forma directa de harcelo, pero no es complicado hacerlo con un poco de código:

```
#!/bin/bash
# array-sort : Sort an array
a=(f e d c b a)
echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
```

```
echo "Sorted array: ${a_sorted[@]}"
```

Cuando lo ejecutamos, el script produce esto:

```
[me@linuxbox ~]$ array-sort
Original array: f e d c b a
Sorted array: a b c d e f
```

El script opera copiando el contenido del array original (`a`) en un segundo array (`a_sorted`) con un un pequeño truco con sustitución de comandos. Esta técnica básica puede usarse para realizar muchos tipos de operaciones en el array cambiando el diseño del entubado.

Borrando un array

Para borrar un array, usa el comando `unset`:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}
[me@linuxbox ~]$
```

`unset` puede usarse también para borrar elementos individuales del array:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset 'foo[2]'
[me@linuxbox ~]$ echo ${foo[@]}
a b d e f
```

En este ejemplo, borramos el tercer elemento del array, subscript 2. Recuerda, los arrays comienzan con el subscript cero, ¡no uno! Fíjate también que el elemento del array debe entrecomillarse para evitar que el shell realice expansión de rutas.

Es interesante cómo, la asignación de un valor vacío a un array no vacía su contenido:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo[@]}
b c d e f
```

Cualquier referencia a una variable array sin un subscript se refiere al elemento cero del array:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ foo=A
[me@linuxbox ~]$ echo ${foo[@]}

A b c d e f
```

Arrays asociativos

Versiones recientes de `bash` soportan ahora *arrays asociativos*. Los arrays asociativos usan cadenas en lugar de enteros como índices del array. Esta capacidad permite nuevos enfoques interesantes en el manejo de datos. Por ejemplo, podemos crear un array llamado "colors" y usar nombres de colores como índices:

```
declare -A colors
colors["red"]="#ff0000"
colors["green"]="#00ff00"
colors["blue"]="#0000ff"
```

Al contrario de los arrays indexados con enteros, que se crean simplemente referenciándolos, los arrays asociativos deben crearse con el comando `declare` usando la nueva opción `-A`. Los elementos de arrays asociativos son accesibles de forma muy parecida a los arrays indexados por enteros:

```
echo ${colors["blue"]}
```

En el próximo capítulo, veremos un script que hace un buen uso de arrays asociativos para producir un interesante informe.

Resumiendo

Si buscamos en la man page de `bash` la palabra "array", encontramos muchas instancias donde bash hace uso de variables array. Muchas de ellas son algo confusas, pero pueden ofrecer una utilidad ocasional en algunas circunstancias especiales. De hecho, todo el tema de los arrays está algo infrautilizado en la programación shell más allá del hecho de que los programas del shell Unix tradicional (como `sh`) carecen de soporte para arrays. Es una desafortunada falta de popularidad ya que los arrays se usan ampliamente en otros lenguajes de programación y proporcionan una herramienta poderosa para resolver muchos tipos de problemas de programación.

Los arrays y los bucles tiene una afinidad popular y a menudo se usan juntos. El formato de bucle

```
for ((expr; expr; expr))
```

está particularmente bien adecuado para calcular subscripts de arrays.

Para saber más

Un par de artículos de Wikipedia sobre las estructuras de datos vistas en este capítulo:

http://en.wikipedia.org/wiki/Scalar_(computing)

http://en.wikipedia.org/wiki/Associative_array

Cosas exóticas

En este, el último capítulo de nuestro viaje, veremos algunos flecos. Aunque hemos cubierto mucho terreno en los capítulos anteriores, hay muchas características de `bash` que no hemos cubierto.

Muchas son algo confusas, y útiles principalmente para los que integran `bash` en una distribución Linux. Sin embargo, hay unas pocas que, aunque no son de uso común, son de ayuda para algunos problemas de programación. Las veremos aquí.

Comandos agrupados y subshells

`bash` permite agrupar comandos. Esto puede hacerse de dos formas; con un *comando agrupado* o con un *subshell*. Aquí tenemos ejemplos de la sintáxis de cada uno de ellos:

Grupo de comandos:

```
{ comando1; comando2; [comando3; ...] }
```

Subshell:

```
(comando1; comando2; [comando3;...])
```

Las dos formas difieren en que el grupo de comandos rodea sus comandos con llaves y el subshell usa paréntesis. Es importante fijarse en que, debido a la forma en que `bash` implementa los grupos de comandos, las llaves deben separarse de los comandos por un espacio y el último comando debe terminar con un punto y coma o con una nueva línea antes de la llave de cierre.

Entonces ¿para qué sirven los grupos de comandos y los subshells? Aunque tienen una diferencia importante (que veremos en un momento), ambos se usan para gestionar redirecciones. Consideremos un segmento de script que realiza redirecciones. Consideremos un segmento de script que realiza redirecciones en múltiples comandos:

```
ls -l > output.txt
echo "Listing of foo.txt" >> output.txt
cat foo.txt >> output.txt
```

Esto es bastante directo. Tres comandos con su salida redireccionada a un archivo llamado `output.txt`. Usando un grupo de comandos, podríamos codificarlo de la siguiente forma:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

Usando un subshell es similar:

```
(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

Usando esta técnica nos hemos ahorrado algo de escritura, pero donde brilla un comando agrupado o un subshell realmente es en los entubados. Cuando construimos un entubado de comandos, es útil a menudo combinar el resultado de varios comandos en una única cadena. Los comandos agrupados y los subshell hacen esto de forma fácil:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

Aquí hemos combinado la salida de nuestros tres comandos y la hemos entubado dentro de la salida de `lpr` para producir un informe impreso.

En el script que sigue, usaremos comandos agrupados y veremos varias técnicas de programación que pueden emplearse junto con arrays asociativos. Este script, llamado `array-2`, cuando se el da

el nombre de un directorio, imprime una lista de archivos en el directorio junto a los nombres de los propietarios de los archivos y de los grupos de los propietario. Al final del listado, el script imprime un listado del número de archivos que pertenecen a cada usuario y grupo. Aquí vemos el resultados (resumidos para abreviar) cuando se le da al script el directorio `/usr/bin`:

```
[me@linuxbox ~]$ array-2 /usr/bin
/usr/bin/2to3-2.6 root root
/usr/bin/2to3 root root
/usr/bin/a2p root root
/usr/bin/abrowser root root
/usr/bin/aconnect root root
/usr/bin/acpi_fakekey root root
/usr/bin/acpi_listen root root
/usr/bin/add-apt-repository root root
.
..
/usr/bin/zipgrep root root
/usr/bin/zipinfo root root
/usr/bin/zipnote root root
/usr/bin/zip root root
/usr/bin/zipsplit root root
/usr/bin/zjsdecode root root
/usr/bin/zsoelim root root
File owners:
daemon : 1 file(s)
root : 1394 file(s)
File group owners:
crontab : 1 file(s)
daemon : 1 file(s)
lpadmin : 1 file(s)
mail : 4 file(s)
mlocate : 1 file(s)
root : 1380 file(s)
shadow : 2 file(s)
ssh : 1 file(s)
tty : 2 file(s)
utmp : 2 file(s)
```

Aquí tenemos un listado (con los números de línea) del script:

```
#!/bin/bash

# array-2: Use arrays to tally file owners

declare -A files file_group file_owner groups owners

if [[ ! -d "$1" ]]; then
echo "Usage: array-2 dir" >&2
exit 1
fi

for i in "$1"/*; do
owner=$(stat -c %U "$i")
```

```
14 group=$(stat -c %G "$i")
15 files["$i"]="$i"
16 file_owner["$i"]=$owner
17 file_group["$i"]=$group
18 ((++owners[$owner]))
19 ((++groups[$group]))
20 done
21
22 # List the collected files
23 { for i in "${files[@]}"; do
24 printf "%-40s %-10s %-10s\n" \
25 "$i" ${file_owner["$i"]} ${file_group["$i"]}
26 done } | sort
```

Echemos un vistazo a la mecánica de este script:

Línea 5: Los arrays asociativos deben crearse con el comando `declare` usando la opción `-A`. en este script creamos los cinco arrays siguientes:

`files` contiene los nombres de los archivos en el directorio, indexados por nombre de archivo
`file_group` contiene el grupo del propietario de cada archivo, indexado por nombre de archivo
`file_owner` contiene el propietario de cada archivo, indexado por nombre de archivo
`group` contiene el número de archivos pertenecientes al grupo indexado
`owner` contiene el número de archivos pertenecientes al propietario indexado

Líneas 7-10: Comprueba para ver que se ha pasado un nombre de directorio válido como parámetro posicional. Si no, se muestra un mensaje de uso y el script sale con un estado de salida de 1.

Líneas 12-20: Hace un bucle a través de los archivos del directorio. Usando el comando `stat`, las lineas 13 y 14 extraen los nombres del propietario del archivo y del grupo del propietario y asigna valores a sus arrays respectivos (líneas 16, 17) usando el nombre del archivo como índice del array. De otra forma el propio nombre del archivo se asigna al array `files` (línea 15).

Líneas 18-19: El número total de archivos pertenecientes al propietario del archivo y al grupo del propietario se incrementan en uno.

Líneas 22-27: Se muestra la lista de archivos. Esto se hace usando la expansión de parámetros "${array[@]}" que se expande en la lista completa de elementos del array cada uno tratado como una palabra separada. Esto permite la posibilidad de que un nombre de archivo contenga espacios en blanco. Fíjate también que el bucle completo está incluido en llaves para que forme un comando agrupado. Esto permite que la salida completa del bucle sea entubada en el comando `sort`. Esto es necesario porque la expansión de los elementos del array no está ordenada.

Líneas 29-40: Estos dos bucles son similares al bucle de la lista de archivos excepto que usan la expansión "${!array[@]}" que se expande en la lista de índices del array en lugar de en la lista de elementos del array.

Sustitución de procesos

Aunque parecen iguales y ambos pueden usarse para combinar cadenas para redireccionarlas, hay una diferencia importante entre los comandos agrupados y lo subshells. Mientras que un comando

agrupado ejecuta todos sus comandos en el shell actual, un subshell (como su nombre indica) ejecuta sus comandos en una copia hijo del shell actual. Esto significa que el entorno se copia y se pasa a una instancia del shell. Cuando el subshell termina, la copia del entorno se pierde, por lo que cualquier cambio hecho al entorno del subshell (incluyendo la asignación de variables) se pierde también. Por lo tanto, en la mayoría de los casos, a menos que un script requiera un subshell, los comandos agrupados son preferibles a los subshells. Los comandos agrupados son también más rápidos y requieren menos memoria.

Vimos un ejemplo del problema del entorno del subshell en el capítulo 28, cuando descubrimos que el comando `read` en un entubado no funciona como esperaríamos intuitivamente. Para resumir, si construimos un entubado como este:

```
echo "foo" | read
echo $REPLY
```

El contenido de la variable `REPLY` siempre está vacío porque el comando `read` se ejecuta en un subshell, y su copia de `REPLY` se destruye cuando el subshell termina.

Como los comandos en entubados siempre se ejecutan en subshells, cualquier comando que asigne variables se encontrará con este problema. Afortunadamente, el shell ofrece una forma exótica de expansión llamada *sustitución de procesos* que puede usarse para solucionar este problema.

La sustitución de procesos se expresa de dos formas:

Para procesos que producen salida estándar:

```
<(lista)
```

o, para procesos que toman entrada estándar:

```
>(lista)
```

donde *lista* es una lista de comandos.

Para resolver nuestro problema con `read`, podemos emplear sustitución de procesos así:

```
read < <(echo "foo")
echo $REPLY
```

La sustitución de procesos nos permite tratar la salida de un subshell como un archivo ordinario para propósitos de redirección. De hecho, como es una forma de expansión, podemos examinar su valor real:

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

Usando `echo` para ver el resultado de la expansión, vemos que la salida del subshell está proporcionada por el archivo llamado `/dev/fd/63`.

La sustitución de procesos se una a menudo con bucles que contienen read. Aquí tenemos un ejemplo de un bucle `read` que procesa el contenido de un listado de directorio creado por un subshell:

```
#!/bin/bash
# pro-sub : demo of process substitution
while read attr links owner group size date time filename; do
cat <<- EOF
Filename: $filename
Size: $size
Owner: $owner
Group: $group
Modified: $date $time
Links: $links
Attributes: $attr
EOF
done < <(ls -l | tail -n +2)
```

El bucle ejecuta `read` para cada línea de un listado de directorio. El propio listado se produce en la línea final del script. Esta línea redirige la salida de la sustitución de procesos en la entrada estándar del bucle. El comando `tail` está incluido en el entubado de la sustitución de procesos para eliminar la primera línea del listado, que no se necesita.

Cuando se ejecuta, el script produce salida como esta:

```
[me@linuxbox ~]$ pro_sub | head -n 20
Filename: addresses.ldif
Size: 14540
Owner: me
Group: me
Modified: 2009-04-02 11:12
Links: 1
Attributes: -rw-r--r--
Filename: bin
Size: 4096
Owner: me
Group: me
Modified: 2009-07-10 07:31
Links: 2
Attributes: drwxr-xr-x
Filename: bookmarks.html
Size: 394213
Owner: me

Group: me
```

Trampas

En el Capítulo 10, vimos cómo, los programas, puede responder a señales. Podemos añadir esta capacidad a nuestros scripts también. Aunque los scripts que hemos escrito hasta ahora no han necesitado esta capacidad (porque tienen tiempos de ejecución muy cortos, y no crean archivos temporales), los scripts más largos y complicados pueden beneficiarse de tener una rutina de manejo de señales.

Cuando diseñamos un script largo y complicado, es importante considerar que ocurre si el usuario cierra la sesión o apaga el ordenador mientras el script se está ejecutando. Cuando ocurre un evento

como este, debe enviarse una señal a todos los procesos afectados. En respuesta, los programas que representan estos procesos pueden realizar acciones para asegurar una terminación apropiada y ordenada del programa. Digamos, por ejemplo, que hemos escrito un script que crea un archivo temporal durante su ejecución. En nombre del buen diseño, habríamos hecho que el script borre el archivo cuando el script termine su trabajo. También sería inteligente hacer que el script borre el archivo si recibe una señal indicando que el programa va a terminar prematuramente.

`bash` ofrece un mecanismo para este propósito conocido como *trampa*. Las trampas se implementan con el apropiadamente denominado comando incluido, `trap`. `trap` usa la siguiente sintaxis:

```
trap argumento señal [señal...]
```

donde *argumento* es una cadena que se leerá y tratará como un comando y *señal* es la especificación de una señal que pone en funcionamiento la ejecución el comando interpretado.

Aquí tenemos un ejemplo simple:

```
#!/bin/bash
# trap-demo : simple signal handling demo
trap "echo 'I am ignoring you.'" SIGINT SIGTERM
for i in {1..5}; do
echo "Iteration $i of 5"
sleep 5
done
```

Este script define una trampa que ejecutará un comando `echo` cada vez que recibe la señal SIGINT o SIGTERM mientras el script se está ejecutando. La ejecución del programa aparece así cuando el usuario intenta detener el script presionando `Ctrl-c`:

```
[me@linuxbox ~]$ trap-demo
Iteration 1 of 5
Iteration 2 of 5
I am ignoring you.
Iteration 3 of 5
I am ignoring you.
Iteration 4 of 5
Iteration 5 of 5
```

Como podemos ver, cada vez que el usuario intenta interrumpir el programa, se muestra el mensaje en su lugar.

Construir una cadena para formar una secuencia útil de comandos puede ser complicado, por lo que es una práctica habitual especificar una función de shell como comandos. En este ejemplo, se especifica una función de shell separada para manejar cada señal:

```
#!/bin/bash
# trap-demo2 : simple signal handling demo
exit_on_signal_SIGINT () {
echo "Script interrupted." 2>&1
exit 0
}
```

```
exit_on_signal_SIGTERM () {
echo "Script terminated." 2>&1
exit 0
}
trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM
for i in {1..5}; do
echo "Iteration $i of 5"
sleep 5
done
```

Este script presenta dos comandos `trap`, uno para cada señal. Cada trampa, por turnos, especifica una función de shell a ejecutar cuando se recibe una señal en particular. Fíjate en la inclusión de un comando `exit` en cada una de las señales de manejo de señales. Si un `exit`, el script continuaría tras completar la función.

Cuando el usuario presione `Ctrl-c` durante la ejecución de este script, el resultado aparece así:

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5

Script interrupted.
```

Archivos temporales

Una razón por la que los gestores de señales se incluyen en los scripts es para eliminar los archivos temporales que el script pueda crear para manejar resultados intermedios durante su ejecución. Hay algo artístico en la denominación de los archivos temporales. Tradicionalmente, los programas en sistemas como-Unix crean sus archivos temporales en el directorio `/tmp`, un directorio compartido creado para tales archivos. Sin embargo, como el directorio está compartido, esto conlleva algunos problemas de seguridad, particularmente para programas que se ejecutan con privilegios de superusuario. Mas allá del paso obvio de establecer permisos apropiados para los archivos expuestos a todos los usuarios del sistema, es importante dar a los archivos temporales nombres no predecibles. Esto evita un exploit conocido como *temp race attack*. Una forma de crear un nombre no predecible (pero descriptivo) es hacer algo como esto:

```
archivotemporal=/tmp($(nombrebase $0).$$.$RANDOM
```

Esto creará un archivo consistente en el nombre del programa, seguido por su ID de proceso (PID), seguido por un entero aleatorio. Fíjate, sin embargo, que la variable de shell `$RANDOM` solo devuelve un valor del rango 1-32767, que no es un rango muy grande en términos informáticos, por lo que una única instancia no es suficiente para vencer a un posible atacante.

Una forma mejor es usar el programa `mktemp` (no confundir con la función de librería estándar `mktemp`) para crear y nombrar el archivo temporal. El progrma `mktemp` acepta una plantilla como argumento que se usa para construir el nombre del archivo. La plantilla debe incluir una serie de "X" caracteres, que se reemplazan con un número correspondiente de letras y números aleatorios. Cuanto más larga sea la serie de "X" caracteres, más larga será la serie de caracteres aleatorios. Aquí tenemos un ejemplo:

```
archivotemporal=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
```

Esto crea un archivo temporal y asigna su nombre a la variable `tempfile`. Los "X" caracteres en la plantilla se reemplazan con letras y números aleatorios de forma que el nombre del archivo final (que, en este ejemplo, también incluye el valor expandido del parámetro especial `$$` para obtener el PID) debería ser algo así:

```
/tmp/foobar. 6593,UOZuvM6654
```

Para scripts que se ejecutan por usuarios normales, sería prudente evitar el uso del directorio `/tmp` y crear un directorio para archivos temporales dentro del directorio home del usuario, con una línea de código como esto:

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```

Ejecución asíncrona

A veces es preferible realizar más de una tarea al mismo tiempo. Hemos visto como los sistemas operativos modernos son al menos multitarea incluso multiusuario. Los script pueden construirse para comportarse de forma multitarea.

Normalmente, esto implica arrancar un script, y por turnos, arrancar uno o más script hijos que realicen una tarea adicional mientras que el script padre continua ejecutando. Sin embargo, cuando una serie de scripts se ejecutan de esta forma, puede haber problemas en la coordinación entre el script padre y los hijos. O sea, ¿qué pasa si el padre o el hijo son dependientes el uno del otro, y un script debe esperar a que el otro termine su tarea ante de finalizar la suya propia?

`bash` tiene un comando interno para ayudarnos a manejar *ejecución asíncrona* como esta. El comando `wait` hace que un script padre se pause hasta que un proceso especificado (p.ej., el script hijo) termine.

wait

Probaremos el comando `wait` primero. Para hacerlo, necesitaremos dos scripts. un script padre:

```
#!/bin/bash
# async-parent : Asynchronous execution demo (parent)
echo "Parent: starting..."
echo "Parent: launching child script..." async-child & pid=$! echo
"Parent: child (PID= $pid) launched."
echo "Parent: continuing..." sleep 2
echo "Parent: pausing to wait for child to finish..." wait $pid
echo "Parent: child is finished. Continuing..." echo "Parent:
parent is done. Exiting."
```

y un script hijo:

```
#!/bin/bash
# async-child : Asynchronous execution demo (child)
echo "Child: child is running..." sleep 5 echo "Child: child is
done. Exiting."
```

En este ejemplo, vemos que el script hijo es muy simple. La acción real la realiza el padre. En el script padre, se arranca el script hijo y se envía al fondo. El ID de proceso del script hijo se graba asignando a la variable `pid` el valor del parámetro de shell `$!`, que siempre cotendrá el ID de proceso del último trabajo puesto en el fondo.

El script padre continua y luego ejecuta un comando `wait` con el PID del proceso hijo. Esto hace que el script padre se pause hasta que el script hijo salga, punto en el cual el script padre concluye.

Cuando se ejecutan, los script padre e hijo producen la siguiente salida:

```
[me@linuxbox ~]$ async-parent
Parent: starting...
Parent: launching child script...
Parent: child (PID= 6741) launched.
Parent: continuing...
Child: child is running...
Parent: pausing to wait for child to finish...
Child: child is done. Exiting.
Parent: child is finished. Continuing...

Parent: parent is done. Exiting.
```

Entubados con nombre

En la mayoría de los sistemas como-Unix, es posible crear un tipo especial de archivo llamado un *entubado con nombre*. Los entubados con nombre se usan para crear una conexión entre dos procesos y pueden usarse igual que otros tipos de archivo. No son muy populares, pero es bueno conocerlos.

Hay una arquitectura de programación común llamada *cliente-servidor*, que puede hacer uso de un método de comunicación como los entubados con nombre, así como de otros tipos de *comunicación entre procesos* tales como conexiones de red.

El tipo más ampliamente usado de sistema cliente-servidor es, claramente, la comunicación entre un navegador web y un servidor web. El navegador web actua como cliente, realizando peticiones al servidor y el servidor responde al navegador con páginas web.

Los entubados con nombre se comportan como archivos, pero en realidad forman buffers "el primero en entrar es el primero en salir" (FIFO - first in firt out). Igual que los entubados normales (sin nombre), los datos entran por un extremo y salen por el otro. Con los entubados con nombre, es posible configurar algo como esto:

```
proceso1 > entubado_con_nombre
```

y

```
proceso2 < entubado_con_nombre
```

y se comportará como si fuera:

proceso1 | *proceso2*

Configurando un entubado con nombre

Primero, debemos crear un entubado con nombre. Esto se hace usando el comando `mkfifo`:

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me me 0 2009-07-17 06:41 pipe1
```

Aquí hemos usado `mkinfo` para crear un entubado con nombre llamado `pipe1`. Usando `ls`, examinamos el archivo y vemos que la primera letra en el campo atributos es "p", indicando que es un entubado con nombre.

Usando entubados con nombre

Para demostrar como funciona el entubado con nombre, necesitaremos dos ventanas de terminal (o alternativamente, dos consolas virtuales). En el primer terminal, introducimos un comando simple y redirigimos su salida al entubado con nombre:

```
[me@linuxbox ~]$ ls -l > pipe1
```

Tras pulsar la tecla `Enter`, el comando parecerá que se ha colgado. Esto es porque no está recibiendo nada desde el otro extremo del entubado aún. Cuando ocurre esto, se dice que el entubado está *bloqueado*. Esta condición se aclarará una vez que apliquemos un proceso al otro extremo y empiece a leer entrada desde el entubado. Usando la segunda ventana de terminal, introducimos este comando:

```
[me@linuxbox ~]$ cat < pipe1
```

y el listado de directorio producido desde la primera ventana de terminal aparece en el segundo terminal como salida desde el comando `cat`. El comando `ls` en el primer terminal se completa con éxito una vez que ya no está bloqueado.

Resumiendo

Bien, hemos completado nuestro viaje. Lo único que queda por hacer ahora es practicar, practicar y practicar. Aunque hemos cubierto bastante terreno en nuestra excursión, apenas hemos rozado la superficie de lo que es la línea de comandos. Hay todavía miles de programas de línea de comandos pendientes de descubrir y disfrutar. ¡Comienza excavando en `/usr/bin` y veras!

Para saber más

- La sección "Comandos compuestos" de la man page de bash contiene una descripción completa de los comandos agrupados y anotaciones sobre subshell.
- Las sección EXPANSIÓN de la man page de bash contiene una subsección de sustitución de procesos.
- La *Guía Avanzada de Scripting Bash* tiene también un apartado sobre sustiución de procesos:
  http://tldp.org/LDP/abs/html/process-sub.html

- El *Linux Journal* tiene dos buenos artículos sobre entubados con nombre. El primero de Septiembre de 1997:
  http://www.linuxjournal.com/article/2156
- y el segundo de Marzo de 2009:
  http://www.linuxjournal.com/content/using-named-pipes-fifos-bash